usenix ASSOCIATION

10th International Conference on Autonomic Computing

San Jose, CA, USA *June 26–28, 2013*

Sponsored by
usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
in cooperation with
ACM SIGARCH

conference

proceedings

10th International Conference on Autonomic Computing (ICAC '13)

San Jose, CA, USA
June 26–28, 2013

Thanks to Our ICAC '13 Sponsors

Silver Sponsor

vmware

Bronze Sponsor

hp

General Sponsor

Google

Media Sponsors and Industry Partners

The Data Center Journal
Distributed Management Task Force (DMTF)
Free Software Magazine
HPCwire
InfoSec News
LXer
No Starch Press
Server Fault
UserFriendly.org

Thanks to Our USENIX and LISA Supporters

USENIX Patrons

Google InfoSys Microsoft Research NetApp VMware

USENIX Benefactors

Akamai EMC Hewlett-Packard *Linux Journal*
Linux Pro Magazine Oracle Puppet Labs

USENIX Partners

Nutanix

USENIX and LISA Partners

Cambridge Computer Google Meraki

ISBN 978-1-931971-02-7

USENIX Association

Proceedings of ICAC ’13: 10th International Conference on Autonomic Computing

June 26–28, 2013
San Jose, CA

Conference Organizers

External Reviewers

ICAC '13:
10th International Conference on Autonomic Computing
June 26–28, 2013
San Jose, CA

Message from the Program Co-Chairs

Welcome to the 2013 International Conference on Autonomic Computing! This year marks the 10th anniversary of ICAC since its inception in 2004. As it has for the past decade, ICAC continues to receive a great deal of interest from researchers and practitioners in the international community. In summary, we received a total of 90 submissions, a 34% increase over the submissions from last year. The increased submission was partly due to the effort to expand the scope of the conference by the addition of two special tracks, Management of Big Data Systems and Self-Aware Internet of Things, which together attracted 17 submissions. Authors were from both academic and industrial institutions in about a dozen countries spread over four continents.

We had a great program committee: a total of 47 members, 31 from academia and 16 from industry. PC members were allowed to submit papers, and there were no submissions from the two PC co-chairs. The committee worked dilligently in the review process, which consisted of three phases. Phase 1: Each PC member was assigned 7–8 reviews, generating a total of 345 reviews, for an average of 4–5 reviews per paper. Around 50 external reviewers offered additional assistance in the initial review phase. Phase 2: The PC members exchanged opinions using HotCRP during a 10-day online discussion phase, mostly focusing on those papers with divergent ratings. Phase 3: The committee had a virtual PC meeting on April 10th via WebEx and teleconference to make final recommendations for the papers that were still under debate. During the meeting, shepherds were assigned to five of the papers to ensure reviewer concerns were addressed in the final papers. Authors and reviewers are who define the research community for ICAC, and we thank you for your contributions to this year's technical program.

From the 73 papers submitted to the main track, the program committee selected 16 full papers, with a 22% acceptance rate. In addition, 10 short papers were accepted, and one was recommended for the MBDS track. These papers cover autonomic computing theories, techniques, and deployments across a variety of systems and application domains, including data centers, clouds, hardware architecture, software-defined networks, and mobile environments. These papers will be presented in nine technical sessions. From the 17 papers submitted to the two special tracks, four papers were accepted to each of the tracks, adding two additional sessions. We also attempted to include both papers with mature results and thorough evaluations as well as early-stage papers that propose new concepts/problems or reach into new areas. In addition, ICAC '13 will offer daily keynote speeches from Carl Waldspurger, Alon Halevy, and Dilma Da Silva, a panel on Big Data Systems, a poster and demo session, and a Ph.D. forum.

We would also like to thank the following organizations and systems that helped us along the way. First, thanks to USENIX for sponsoring ICAC and hosting it as part of the Federated Conference Week this year. The USENIX staff has been very helpful and supportive during the whole process; without them our jobs would have been much harder. Second, we thank our corporate sponsors and partners, including VMware (Silver Sponsor), HP (Bronze Sponsor), Google (General Sponsor), SPEC Research, and DMTF. Third, we appreciate the variety of features offered by the HotCRP system that significantly reduced the overhead of running the PC. Last but not least, we want to thank everyone else in the ICAC '13 organizing committee, who helped put together a strong technical program. Thank you for your involvement and participation in the ICAC community, and enjoy the conference!

Calton Pu, ***Georgia Institute of Technology***
Xiaoyun Zhu, ***VMware***
ICAC '13 Program Co-Chairs

Application Placement and Demand Distribution in a Global Elastic Cloud: A Unified Approach

Hangwei Qian
VMWare
Palo Alto, CA, 94304

Michael Rabinovich
Case Western Reserve University
Cleveland, OH 44106

Abstract

Efficient hosting of applications in a globally distributed multi-tenant cloud computing platform requires policies to decide where to place application replicas and how to distribute client requests among these replicas in response to the dynamic demand. We present a unified method that computes both policies together based on a sequence of min-cost flow models. Further, since optimization problems are generally very large-scale in this environment, we propose a novel demand clustering approach to make them computationally practical. An experimental evaluation, both through large-scale simulation and a prototype in a testbed deployment, shows significant promise of our approach for the targeted environment.

1 Introduction

An important benefit of cloud computing is that it allows Internet application providers to obtain global footprint and elastic capacity without the need to deploy and maintain their own infrastructure. This service is often referred to as IaaS ("Infrastructure as a Service"). Cloud providers can offer IaaS efficiently by deriving the economy of scale though multiplexing their shared platforms among multiple applications. A number of cloud providers, including Google, Microsoft, and Amazon, offer some variation of this capability.

These geo-distributed multi-tenant hosting platforms need to be able to effectively distribute the hosted applications across multiple data centers and direct client demand to the appropriate application replicas. Specifically, this task involves the following two key policies: (i) At how many and which data centers should each application be deployed? We refer to this as the (global) application placement problem; and (ii) How should client demand be distributed to these application replicas? This is commonly referred to as demand distribution or, interchangeably, server selection problem.

Much prior work has targeted environments addressing one or the other of these aspects (see § 8). However, an elastic cloud must deal with both issues simultaneously because it distributes demand among dynamically changing sets of application instances. Consequently, we propose a unified framework to compute these two policies simultaneously. A comparison with an existing approach that also addressed both policies but computed them in isolation showed a significant advantage of our approach (§ 6.6).

Computing these policies in a hosting cloud brings an additional challenge. Because request processing involves accesses of application-specific back-end servers, the proximity of a request to data centers depends not just on the client's location but also on the location of the back-end servers and hence on the requested application. This increases the scale of the optimization problems by orders of magnitude (§ 4). We propose a novel demand clustering approach we call *permutation prefix clustering*, and show that it makes global optimization practical in many environments.

In summary, this work addresses the application placement and demand distribution problems in a geo-distributed and globally-shared cloud platform, and makes the following contributions: (i) We propose and evaluate a unified framework to jointly solve the application placement and demand distribution problems; (ii) A novel clustering technique is introduced to scale our optimization model to realistic platform sizes, which we believe will prove valuable for other optimization models as well; and (iii) We prototype our approach and demonstrate its operation in a testbed deployment.

2 System Overview

The high-level view of our targeted environment is shown in Fig. 1. Each client connects to a request-routing component (e.g., DNS server or HTTP redirector), which directs it to a data center hosting requested application. Known mechanisms (such as one provided in WebLogic [1]) ensure continued session state availability even if a client is redirected to a new instance mid-session. When processing requests, the application is assumed to access back-end database located at the premises of the application providers for security or legal reasons (the extended version of this paper also considers the hosted database scenario [27]). Thus, we aggregate the network distances from clients to data centers and from data centers to databases when calculating the network delay for the requests. Note, obtaining the distance information efficiently is a complicated task and an important part of the providers' know-how. We assume this information is supplied by a separate measurement component (not considered here).

To effectively manage the proximity information of all the Internet clients, we assume the platform groups its clients by the IP prefixes found in multiple BGP tables [19], with each group dubbed a *client cluster (CC)*. A key part of our work is to aggregate demand further so that computing the application placement and demand distribution policies becomes tractable. Also, many cloud providers concentrate their platforms in a small number of strategically located mega data centers, leading to the factors of 5 to 7 decrease in the operational cost [12]. For example, Amazon EC2 is deployed in four locations (US-East, US-West, Ireland, and Singapore); other infrastructure providers, such as Limelight and AT&T, have a couple dozens data centers. We assume roughly this number of data centers (around 20). We do not target platforms such as Akamai with presence in thousands of locations.

We focus on the business models (e.g., *auto-scaling* option in EC2 or Google's AppEngine), in which the cloud itself makes the decisions on the number and location of various application instances, to maximize the application performance and minimize the number of data centers where these applications are deployed. Removing underutilized application instances reduces the customer costs and frees up resources for other applications. Another objective is to reduce the number of placement changes in consecutive configurations. Despite recent advances in reducing overhead of starting a virtual machine [21] or an application server [9], deploying an application instance remains a heavy-weight operation in terms of CPU costs and system reconfiguration.

In making placement decisions, we only consider whether or not an application is deployed in a data center. Others have addressed the problem of resource allocation among applications within a data center [37, 26, 33]. In the rest of the paper, an "application instance" means that the application is deployed at the data center, regardless of the amount of resources it is assigned locally.

Resource allocation decisions require monitoring the demand and utilization of data centers. We assume a central controller collects this information periodically from each data center. Like any platform based on request routing, our target environment requires translation between requests and service demands; this so-called *application modeling* problem has been studied intensively (e.g., [32, 34, 35]) and we assume the use of one of these existing technologies. We also assume that our applications (i.e., web sites) are sufficiently popular so that even if different requests to a web site have different service demands, for a reasonable request rate (e.g., higher than the deletion threshold - see § 5), these requests will result in a representative request mix. (If this assumption does not hold for an application, its requests must be split into classes with similar service demands and each class modeled separately.) Meanwhile, request rates for different applications are normalized so that the same (normalized) request rate will result in the same resource utilization regardless of the application. Thus, a request rate translates to the proportional resource usage and can be used to measure the capacity and the utilization of data centers. (This assumption is supported by our experience with prototype in § 7.)

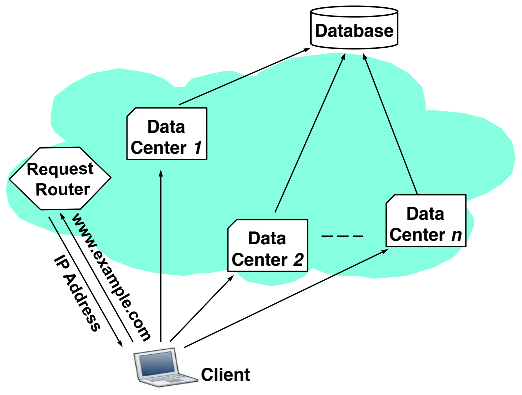


Figure 1: Overview

2.1 Problem Statement

Let D be the number of data centers, A the number of applications and C the number of client clusters. The placement policy can be described as an $A \times D$ matrix P, with element $P_{ij} = 1$ if application i is deployed at data center j; $P_{ij} = 0$, otherwise. The demand distribution policy is an $A \times C \times D$ matrix R, whose element R_{amn} is the fraction of requests from client cluster m for application a to be directed to data center n. The system enacts the distribution policy by directing a request from client cluster m for application a to data center n with probability R_{amn}. Let r_{am} be the request rate for application a from client cluster m. Assume each request is associated with a cost C_{amn} if it is served at data center n, and u_n is the utilization of the data center. We formulate our problem as a *multi-objective optimization problem* [23] fulfilling the following competing objectives:

$$Minimize \sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} \sum_{n=1}^{D} R_{amn} C_{amn} \quad (1)$$

$$Minimize \sum_{a=1}^{A} \sum_{n=1}^{D} P_{an} \quad (2)$$

$$Minimize \sum_{a=1}^{A} \sum_{n=1}^{D} |P_{an} - P_{an}^{prev}| \quad (3)$$

subject to

$$\sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} R_{amn} \leq u_n, n = 1, 2, \ldots, D \quad (4)$$

$$0 \le R_{amn} \le 1; \sum_{n=1}^{D} R_{amn} = 1 \quad (5)$$

$$P_{an} \in \{0,1\};\ R_{amn} > 0 \ implies \ P_{an} = 1 \quad (6)$$

where P_{an}^{prev} is the previous placement policy. Objective (1) minimizes the overall cost. While C_{amn} is an abstract cost function, we use aggregate distance (measured as network latency) as the cost function thus trying to minimize the overall user-perceived network latency. Objective (2) minimizes the number of data centers with deployed application replicas and objective (3) minimizes the number of placement changes.

While multi-objective optimization problems are commonly handled by combining all the objectives into a single one with some weights assigned to each objective, in our case, this would transform the problem to a mixed integer programming formulation (in fact, a variant of a multicommodity capacitated facility location problem [8]), which is NP-hard. Further, choosing appropriate weights for different objectives is difficult in our context as their effect on the final policies is indirect and non-intuitive. Instead, we handle the problem heuristically as follows.

2.2 Framework

Our heuristic approach to arbitrate among the competing objectives involves two steps. First, we compute optimal request distribution among data centers assuming every application is deployed at every data center (*full deployment*). Here any optimization technique can be applied. We explore a centralized approach based on a min-cost max-flow model (§ 3).

Second, given the optimal demand distribution policy, we attempt to remove underutilized instances. We introduce a *Deletion Threshold (DT)* as the level of demand that justifies the cost of running an application at a data center (note that the DT can be selected independently for each application and has an easily grasped intuitive meaning). We try to remove instances whose demand after the first step is below DT by reassigning their flows to remaining instances in an optimal manner (§ 5). We also attempt to reduce the number of placement changes in this step by assigning lower deletion threshold to already-deployed instances (§ 5.3).

3 Full Deployment

We begin by obtaining optimal demand distribution policy with full deployment. We use a min-cost max-flow optimization model for this purpose. This model represents the system as a directed network, with *source nodes* generating demand, *sink nodes* consuming this demand, and demand flowing from sources to sinks along edges labeled with $(cost, capacity)$. An edge label indicates the maximum amount of demand that can traverse this edge and the unit cost of such traversal. There are efficient algorithms that solve the min-cost max-flow problem, i.e., find the assignment of demand to edges that maximizes the total satisfied demand while minimizing the total cost. Refer to [7] for details on *min-cost* flow problem and [2] for transforming it to a *min-cost max-flow* problem; we use both terms interchangeably. We utilize the tool [4] in our implementation, which uses an algorithm with complexity $O(V^2 Elog(VCap))$ [16] where V and E are the number of nodes and edges and *Cap* is the maximum edge capacity.

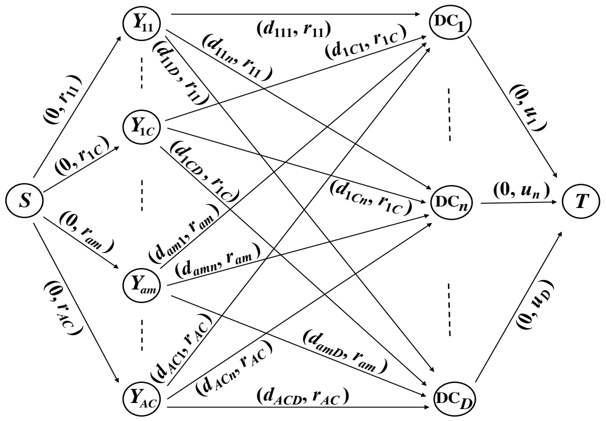


Figure 2: Min-cost network model

3.1 Problem Modeling

We would like to forward client requests to closest data centers and at the same time avoid overloading any data centers. We assume the service does not degrade appreciably as long as data center utilization is below its capacity. (In reality, this means that utilization must stay below a certain *watermark*, which for now we view as capacity but set as a parameter in the simulation – see § 6.) Under this notion, we model our problem as the following min-cost flow network.

Because of different back-end servers, requests for different applications from the same client may have different aggregate distances to the same data center. Thus our model can not simply consider all demand from the same client cluster as a whole. Therefore, as shown in Figure 2, we have a pair-node $Y_{am}, a = 1, 2, ...A, m = 1, 2, ...C$ for each application and client cluster pair (a, m). Also, each data center n has a node DC_n. Finally, we have a source node S and sink node T. From source S to each pair-node Y_{am}, we add an edge with cost 0 and capacity r_{am}, the latter being the request rate from client cluster m for application a. Then we add an edge from each pair-node Y_{am} to each data center node DC_n, with cost being the aggregate distance d_{amn} when client cluster m accesses the application a at data center n, and capacity equal to the full request rate from this client cluster for this application (since the actual data center capacity is enforced by the subsequent edge), i.e., r_{am}. By connecting each pair-node with every data center, we allow the demand from the corresponding client cluster to be potentially split among any of the data centers. Finally, there is an edge from every data center node DC_n to sink T, with cost 0 and capacity equal to the capacity of data center u_n.

We try to move the total amount of flow $\sum_{a=1}^{A}\sum_{m=1}^{C} r_{am}$ from source S to sink T with minimum cost. After we obtain the solution, flow f_{amn} on the edge between nodes Y_{am} and DC_n represents the amount of requests from client cluster m for application a that should be forwarded to data center n.

4 Permutation Prefix Clustering

The size of the min-cost flow problem in Fig. 2 is extremely large, with $A \cdot C + D + 2$ nodes and $A \cdot C + A \cdot C \cdot D + D$ edges. According to [19], there were on the order of 400,000 client clusters in 2000. Then, for $C = 400,000$ client clusters, $A = 100$ applications, and $D = 20$ data centers, the number of nodes and edges are in the order of 4×10^7 and 8×10^8 respectively, making this problem intractable. We address the scalability problem in this section.

4.1 Basic Idea

With aggregate distance, each pair-node Y_{am} has its own preference of data centers in terms of proximity, producing a permutation of data centers. For example, permutation {1,4,2,3,6,5} means requests for application a from client cluster m are the closest to DC_1, the second closest to DC_4, and so on. We define each permutation as a *region*, and client requests with the same preference of data centers are in the same region. There is a region for each pair-node in Fig. 2. We propose *permutation prefix clustering* to reduce the number of regions and thus the number of edges in Fig. 2.

In this method, we merge regions if their permutations share the same prefix of certain length. For example, for six data centers, let $region_1$ have permutation {1,4,2,3,6,5} and $region_2$ have permutation {1,4,2,3,5,6}. With prefix length 4, we could merge them into $region_{12}$ with prefix {1,4,2,3}. (Note that requests from the *same* client cluster for different applications may end up in *different* regions since their data center preferences may be different due to different back-end servers.) After merging, we compute the distance from the new region to each data center, including those beyond the prefix, as the weighted average of the distances from $region_1$ and $region_2$, with request rate from each region as the weight.

Our observation is that unless most data centers are highly loaded, requests for an application will only go

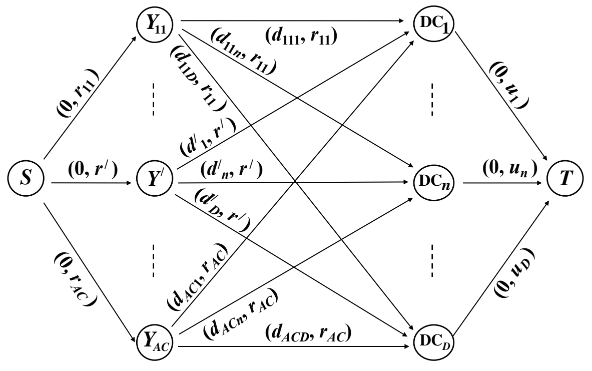


Figure 3: Clustered network model

to a few closest data centers. So for each client request, we only need to consider the front part of its corresponding permutation. Admittedly, there would be proximity penalty when the flows do need to go to the data centers beyond the prefix. However, this happens when most data centers are highly loaded, in which case the proximity becomes less of a priority as we need to satisfy all the demand first. Moreover, our use of the weighted average distance to *all* data centers, including those beyond the prefix, significantly reduces this penalty (see § 6.3).

4.2 Application to Min-Cost Model

To illustrate how *permutation prefix clustering* is applied in our min-cost flow model, suppose we want to merge the regions for pair-nodes Y_{1C} and Y_{am} in Fig. 2 because their permutations share a prefix. We remove nodes Y_{1C} and Y_{am} along with all their adjacent edges and replace them with a new node Y'. An edge is added from source node S to node Y', and from Y' to each node $DC_n, n = 1, 2, ...D$. The cost of the edge from S to Y' is still zero and capacity is the sum of the capacities of the edges (S, Y_{1C}) and (S, Y_{am}), or $r' = r_{1C} + r_{am}$. The cost of the edge $(Y', DC_n), n = 1, 2, ...D$ is the weighted average of cost of edges (Y_{1C}, DC_n) and (Y_{am}, DC_n), or $d'_n = \frac{d_{1Cn} * r_{1C} + d_{amn} * r_{am}}{r_{1C} + r_{am}}$, and capacity is r'. The updated network is shown in Fig. 3. This technique generalizes trivially to merging more than two pair-nodes.

Let L be the length of the permutation prefix. Then the total number of possible regions after merging is:

$$Min\{A * C, \prod_{i=0}^{L-1}(D - i)\}$$

which means the same number of merged pair-nodes Y'. Also, the dominant element of the total number of edges in Fig. 2 is reduced from $A \cdot C \cdot D$ to:

$$D * Min\{A * C, \prod_{i=0}^{L-1}(D - i)\}$$

Since $A * C$ is very large, the total number of nodes and edges in Fig. 3 are in the order of $\prod_{i=0}^{L-1}(D - i)$ and

$D * \prod_{i=0}^{L-1}(D-i)$ respectively, depending on D and L only. Generally, the smaller the prefix length, the smaller the problem size but the larger the potential proximity penalty. We study these effects in § 6.3.

5 Partial Application Placement

A solution to the model of Fig. 3 provides a demand distribution policy assuming full deployment. Our next step is to remove underutilized application instances.

Let f_{an} be the amount of request flow of application a assigned to data center n. If $f_{an} \geq DT$, we call it *normal flow* and keep the instance of application a at data center n. We denote the set of data centers with these instances as U_a. We can immediately remove an instance with zero demand (e.g., if $f_{an} = 0$). The rest of this section handles instances with demand $0 < f_{an} < DT$. We call them *tiny instances* and their flows *tiny flows*.

5.1 Heuristics

Let set $V_a = \{DC_n | 0 < f_{an} < DT\}$ contain data centers with a tiny instance of application a. Also let h_n be the number of normal flows at data center DC_n. We first assume that all tiny instances are removed (unless all instances of an application are tiny, in which case one instance with the largest flow is retained) along with their flows and increase the residual capacities of the affected data centers accordingly. We then attempt to distribute these flows (referred to as residual demand) to data centers with residual capacities. Our procedure is guided by the following observations:

1. We should try to remove the instances with the smallest flows first because the reassignment of small flows will affect fewer requests. In particular, it means that (1a) demand for a tiny instance should not be reassigned to an even tinier instance, and (1b) we should try to accommodate smaller flows (across all applications) first.
2. If we must retain some tiny instances (because data centers in U_a reach their capacity), we should keep the tiny instances with the largest flows first. This is again motivated by the desire to keep the largest amount of demand assigned to the nearest data centers.
3. When selecting data centers in U_a to assign residual demand, we should favor those with smaller h_n because their residual capacity is harder to utilize (since a data center can only accept additional demand for the applications it hosts).

While the above set of heuristics may suggest a simple greedy procedure, where we reassign flows in the increasing size order and distribute them to normal instances first and then to the largest tiny instance with residual capacity, this may result in highly suboptimal flow assignment. Instead, we again build a min-cost flow model for this problem, so that we reassign the residual demand optimally, and at the same time manipulate the costs in the model to follow the above heuristics.

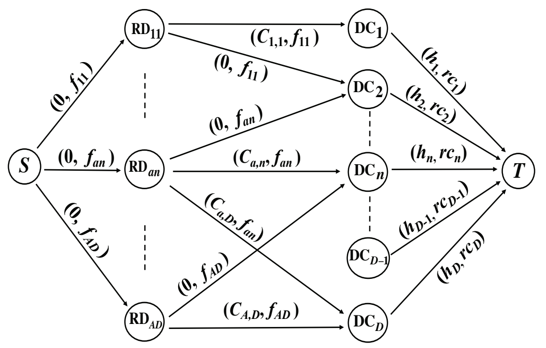


Figure 4: Residual demand distribution network

5.2 Tiny Flow Removal

Our min-cost flow model for tiny flow removal is shown in Fig. 4. Each tiny flow f_{an} has a corresponding node RD_{an}, referred to as *demand node*. From source S, we add an edge to every demand node. Also, from each demand node RD_{an}, there is an edge to data center node DC_k if the latter has an instance of application a and $f_{ak} >= f_{an}$. By not including edges to data centers with smaller flows (note the absence of edges from RD_{an} to DC_1 and DC_{D-1} in Fig. 4), we enforce heuristic 1a. Finally, each data center node is connected to sink T.

For edges from source node S to demand node RD_{an}, the capacity is f_{an}, and the cost is 0 - this represents the demand to be satisfied. All edges from demand node RD_{an} to data center nodes have capacity f_{an} (this demand could potentially be satisfied by any of these data centers), and the edges from data center nodes to the sink have capacities equal to the residual capacity rc_n of each data center. For data center $DC_k \in U_a$, the cost of the edge from node RD_{an} to DC_k is 0 (since it already has an instance and we would like to assign as much demand as possible to these nodes – see the edge from RD_{an} to DC_2 in the figure). The cost of other edges is chosen in a way such that:

1. For any two tiny flows f_{an} and $f_{a'n'}$, if $f_{an} < f_{a'n'}$ then $cost_{i,j}$ of edges going from demand node RD_{an} to data center nodes in V_a is larger than $cost_{i',j'}$ of edges going from $RD_{a'n'}$ to data center nodes in $V_{a'}$. In this way, flow f_{an} would have an advantage over $f_{a'n'}$ when competing for residual capacity of data centers with instances of both applications, thus following heuristic 1b.
2. The cost of edges going from residual node RD_{an} to $DC_k \in V_a$ is inversely proportional to f_{ak}. In this way, the min-cost flow algorithm will try to follow heuristic 2.
3. For the edge from data center node DC_n to sink T, the cost is the number of normal flows h_n at data center DC_n. This makes the algorithm follow heuristic 3.
4. Because heuristics 1 and 2 have higher priority than 3, we make sure that the cost of edges from demand nodes RD_{an} to data center nodes in V_a dominates the cost of edges from data center nodes in V_a to the sink node. In Fig. 4, $C_{a,n} >> h_k$ and $C_{a,D} >> h_k$ for all

$k = 1, 2, ...D$.

After solving this problem, we remove all the tiny instances that became idle (assigned no demand).

5.3 Hysteresis Placement

As described so far, our scheme computes a new placement policy only based on the current demand distribution, regardless of the previous placement. This can result in large number of placement changes.

We propose *hysteresis placement* to control the number of placement updates. We introduce a parameter *hysteresis ratio (HR)* when categorizing flows. If application a is deployed at data center n in the previous placement, we consider f_{an} as tiny instance only when $f_{an} < \frac{DT}{HR}$, where $HR \geq 1$. In this way, if application a is currently deployed at data center n, then it is more likely to be kept in place in the new placement. This added "stickiness" may result in some increase in the number of application instances and some response time penalty. We evaluate these effects in § 6.5.

5.4 Further Fine-Tuning

Our scheme so far aggregates demand into flows from coarse-grained regions and does not distinguish between requests within a flow. So, when a flow is assigned to multiple data centers, rather than sending requests to these data centers at random, we can split this flow among its assigned data centers according to request proximity preferences as long as this does not violate the overall demand distribution. We skip details due to space limitation but provide them in the extended version of this paper [27]. Our evaluation study includes this optimization in all the experiments.

6 Evaluation

We study the performance of our approach using large-scale simulation built on CSIM [6], a discrete-event simulation package. Mimicking the actual system, our simulator has a decision component and a request routing component. The decision component periodically updates application placement and server selection policy (every 30s by default). There is also a workload component that generates requests according to load patterns discussed later. The routing component forwards each request to the appropriate data center according to the policy generated by decision component.

6.1 Cloud Model

We simulate a global cloud platform across 20 data centers hosting 100 applications (except for the scalability experiments in § 6.7). We parameterize our model as follows. We got all pingable IP addresses – 157803 total – from the Gnutella peer list compiled at the University of Oregon [3] and found their geographical locations using the GeoIP database (commercial version)[5]. We "deploy" our 20 data centers in countries according to their client distribution, i.e., nine data centers in US, three in China, etc. For the US, we use a similar procedure to distribute the data centers among states.

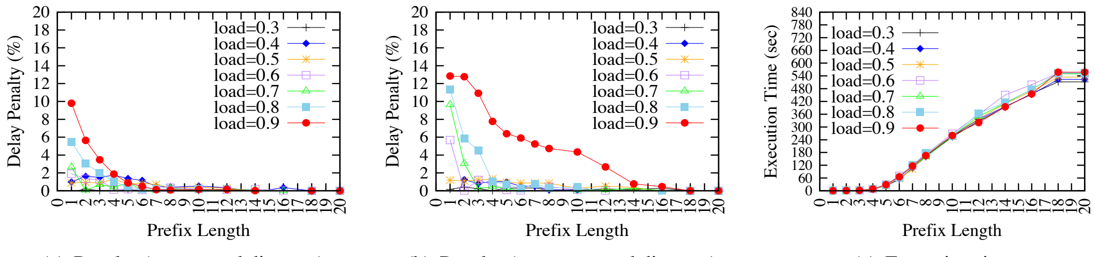
We then selected 20 PlanetLab nodes in the same locales as our data centers and measured ping latencies from each such PlanetLab node to each client. We were able to obtain complete distances to 100546 clients. We then used these clients to represent the locations of client clusters, the 20 PlanetLab nodes to mimic our data centers, and the measured ping latencies as the network distances. Since back-end databases are assumed to stay outside the cloud (see § 2), we randomly select 100 PlanetLab nodes to mimic the databases and use ping latencies from the 20 PlanetLab nodes representing data centers to these 100 PlanetLab nodes as distances between data centers and databases.

We divide the world into 20 geographic regions, each with a data center. Client clusters that share a common closest data center fall into the same geographic region with the data center. We also divide applications into two categories, regional and global. Regional applications are particularly popular within a specific geographic region (*hot region*), e.g., the website of a state government; global applications are universally popular. For a regional application, we define *regional rate* as the portion of requests it receives from its hot region. We use regional rate of 0.9.

6.2 Workload

Each data center can serve 10,000 requests peer second (req/s), resulting in the total capacity of all data centers of 200,000 req/s. These rates are dictated by the scalability of the simulator itself, but are sufficient to evaluate our approach. We define $load\_factor$ as the ratio of the total request rate of all data centers to the total capacity. Each (normalized –see § 2) request is assumed to have service time 0.03 second, so every data center in the simulator has 300 CSIM facilities that mimic servers. We set the queue length of each facility to 150; requests are distributed among servers in a data center in a round-robin fashion and are dropped when arriving at a facility with full queue. In the optimization models, we assume the capacity watermark of 0.9, that is, the system tries to keep each data center utilization within 9,000 req/s.

Demand Generation. We assume applications' popularity follows Zipf law with parameter 1. The top application is global and the remaining 99 are regional, thus the global application generates around 20% of total demand. Given the target total request rate, r, determined by the load factor, the workload generates requests sequentially with exponentially distributed inter-arrival time with mean $t = 1/r$. For each request, it first



(a) Penalty (aggregated distance) (b) Penalty (no aggregated distance) (c) Execution time

Figure 5: Performance of prefix clustering

selects an application according to the power law probability distribution. Then if the selected application is regional, it assigns the request to a random client cluster from its hot region with probability of $regional\_rate$ and to a randomly selected client cluster from outside its hot region otherwise. If the application is global, the request is assigned to a random client cluster.

Dynamic Demand Patterns. During simulations, the demand pattern changes every T seconds. We use the following dynamic load patterns in our experiments:

1) *Vary-All-App:* starting from the initial distribution generated as described above, the demand for each application changes randomly within $\pm\Delta\%$, where Δ is a parameter controlling the extent of variability. This workload is an extended version of vary-all-apps in [33].

2) *Rank-Exchange:* popularity rankings of k randomly picked pairs of applications are swapped, where k controls the extent of demand variability. This workload mimics the change of popularity among applications.

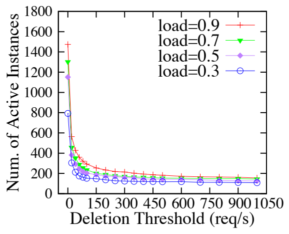
3) *Reshuffle-All:* in each cycle, the rankings of the applications are reset to a random permutation and each regional application is remapped to a new random region. This workload mimics extreme case of change, where the demand pattern in each cycle is completely independent of the pattern in previous cycle.

6.3 Clustering Performance

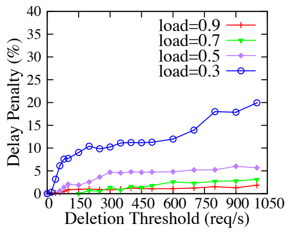
We begin with the evaluation of permutation prefix clustering. In each experiment, we initially generate the requests that would occur in one second and re-send these requests repeatedly every logical second for ten logical seconds, at which point we recompute the demand to be used for the next ten seconds, and so on. While we use the same demand pattern, the demand will be different due to new random coin tosses during generation. To factor out the effects of stale demand data, the experiments in this subsection as well as § 6.4 and § 6.5 recompute the policy every time the demand is recomputed (every 10 second here) and use the upcoming demand data as input. We defer considering online policy computation based on prior demand until policy evaluation and prototype testing (§ 6.6 and § 7). The simulation lasts 50 logical seconds. To concentrate on clustering effect on server selection, all experiments in this subsection assume full deployment for each application, deletion threshold 0 req/s, and hysteresis ratio 1.

Fig. 5 shows performance effects as clustering level changes from the extreme case when only the closest server is considered (prefix 1) to no clustering (prefix 20, although no clustering occurred beyong prefix 18). We measure the number of dropped requests (although we did not observe any) and the average response time. Fig. 5a shows the response time penalty from clustering (also called *delay penalty* below), expressed as the relative difference between average response times with and without clustering. As seen from the figure, for a given level of clustering (i.e., the prefix length value), the penalty is smaller for lower load factors. (The line for load factor 0.4 deviates slightly from this trend for initial values of the prefix length. Since the penalty variations involved are very small - within 1% - we view it as a statistical aberration.) This makes sense because with low load, most demand is satisfied by the closest server, and the discrimination among more distant servers becomes unimportant. However, even for high loads, the clustering penalty is small, never exceeding 10%, and drops quickly with the prefix length. We attribute this to the effect of our distance aggregation for all members of the cluster: even when client-application pairs are clustered, their proximity to servers beyond the common prefix is still accounted for through aggregated distances. Indeed, Fig. 5b shows the delay penalty increases significantly when all distances to data centers beyond the prefix are assumed equal. Finally, Fig. 5c depicts the effect of clustering on the algorithm execution time. It shows that clustering trades these small delay penalties for a dramatic reduction in the execution time. For instance, for load factor 0.6, going from no clustering (prefix 20) to clustering with prefix 3 reduces the execution time from 552.4s to 2.3s, at the expense of only 1.3% delay penalty. We study the scalability of our approach further in § 6.7.

In summary, our experiments show that prefix clustering is a promising general technique for aggregating demand. Given these results, we use prefix size 3 for sub-

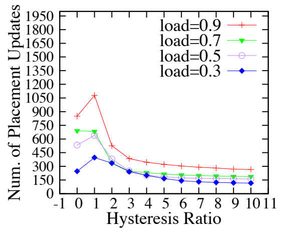


(a) Number of instances

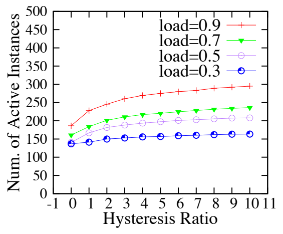


(b) Delay penalty

Figure 6: The effects of the deletion threshold



(a) Placement updates



(b) Number of instances

Figure 7: The effects of the hysteresis ratio

sequent experiments, which allows us to solve the min-cost problem efficiently while keeping the delay penalty small - within 4% in the above experiments.

6.4 Deletion Threshold

We now study the deletion threshold (DT) effect. A higher DT tends to remove more instances but leads to greater performance penalty, as requests that used to go to the underutilized instances will now be routed to more distant data centers, while DT = 0 means no tiny instance removal, i.e., only completely idle instances are dropped. We use prefix 3 (see § 6.3) and hysteresis ratio 1 for these experiments.

Fig. 6 quantifies these effects by showing the total number of instances and delay penalty for different DT values. The workload is the same as in the previous subsection. Since each simulation run involves five recomputations of the demand, each data point represents the average total number of application instances across the whole run. The figure shows that as the deletion threshold increases, the number of total instances plunges in the beginning, but then decreases very slowly. The delay penalty behaves the opposite way, although at low load the penalty does not flatten. In general, this result indicates that with an appropriate deletion threshold, our scheme can drastically reduce the number of application instances with small performance penalty. We choose deletion threshold 150 req/s throughout our subsequent experiments as it obtains factor of 5-7 reduction in the number of application instances while keeping the delay penalty under 8% for all loads.

6.5 Hysteresis Placement Effects

We now evaluate the hysteresis ratio effects, using deletion threshold 150 req/s (see § 6.4) and prefix length 3 (see § 6.3) in the experiments.

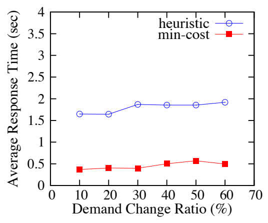
We use the following workload. At the first logical second, we generate a demand. At the next second, we remap the regional applications randomly to regions and recompute the demand. At all the subsequent seconds, we recompute the demand with new random coin tosses but keep the same pattern. So the workload changes dramatically in the second second, but keeps stable (except for statistical variations) in the remaining time. The application placement is computed every second: with our focus on placement changes, this allows us to shorten the experiment without affecting the results. The experiment lasts 20s. In Fig. 7, each data point represents the mean over five simulation runs with different seeds.

Fig. 7a shows the number of placement changes as the hysteresis ratio increases. For comparison, the figure also includes results for a heuristic application placement from [29] at 0 point on the x-axis. We see that with the increase of the hysteresis ratio, the number of placement updates drops but the total number of instances increases. When hysteresis ratio reaches 3, our approach results in *fewer* placement updates than algorithm from [29], even though the latter computes the new placement by adjusting the current configuration. Admittedly, as Fig. 7b shows, this comes at the expense of a certain increase in the number of instances, especially at higher load factors (a third more instances). We argue that this modest increase is justified by a significantly better performance of our approach, as we will see in the § 6.6. Interestingly, the delay penalty is negligible - less than 2% – and is not shown here. We use hysteresis ratio 3 for the rest of our experiments.

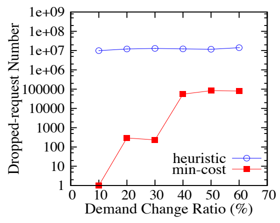
6.6 Policy Evaluation

This section compares the quality of the policies produced by our approach and prior work. To our knowledge, the only works that jointly address the problems of demand distribution and application placement are [29] and [25]. Since our approach and [25] are not directly comparable ([25] aims at minimizing the replica load imbalance rather than optimizing the proximity), we compare our approach with [29]. The latter represents a drastically different approach from ours: it heuristically adjusts current placement by replicating or migrating instances and modifies server selection strategy according to the observed demand.

In these experiments, we use the dynamic load patterns in § 6.2 with load factor 0.5 and regional rate 0.9. Experiments start with full deployment and every request is forwarded to the closest data center. We generate the initial demand according to § 6.2. For the first

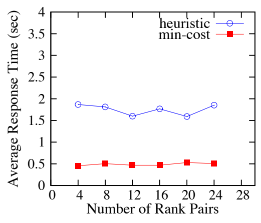


(a) Average response time

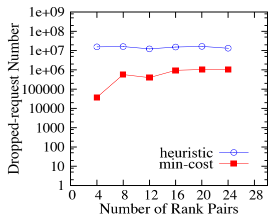


(b) Dropped requests

Figure 8: Policy performance (Vary-All-App)



(a) Average response time



(b) Dropped requests

Figure 9: Policy performance (Exchange-Rank)

15 logical seconds, the system is in a warm-up stage, where we update the policies every second so that the policies reflect the initial demand pattern after this stage. This is done for fairness to [29] as it adopts to the desired configuration incrementally. Then the system goes into the measurement stage, lasting 900 logical seconds, in which the demand is recomputed every 150 sec. according to the dynamic load pattern used, and the policies are updated every 30 sec. When computing the policies, we collect request rates through the statistics from all data centers as in reality. We use exponential moving average (smoothing factor 0.6) to maintain these statistics.

Fig. 8, 9 show the average response time and number of dropped requests for the two approaches, for the first two workloads (Reshuffle-All is not shown due to the space limit and is included in the extended version [27]). The curves corresponding to our approach and the approach of [29] are labeled, respectively, "min-cost" and "heuristic". The results show dramatic performance advantage of our approach for both metrics. The average response time shows improvements at least by a factor of 2, and dropped requests reduce by orders of magnitude.

6.7 Scalability

We turn to the scalability of our approach. Our baseline setup of the system includes 100,546 client clusters, 20 data centers and 100 applications. We measure the execution time of our algorithms by increasing one of these parameters and keeping the other two constant. For the purpose of simulations, whenever we add a new entity to the setup and need a network delay between it and other entities, we pick the delay at random between 0 and 500ms. We utilize Dell PowerEdge 2950 server with 8 cores and 16G memory in the experiments.

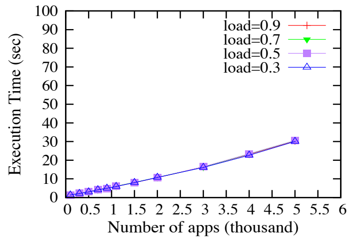
The results are presented in Fig. 10. They show that the execution time grows almost linearly with the number of client clusters and applications, but superlinearly with the number of data centers. The latter makes sense since, for the prefix length 3 we used, the size of the min-cost flow model of Fig. 3 grows as the power of 4 of the number of data centers (§ 4.2). Meanwhile, when the number of client clusters and applications increases, the size of the min-cost flow problem used in the first phase (optimization with full deployment) does not change, but the size of the problems used in the second phase (application placement) and in the flow-splitting phase increase linearly. As shown, the execution time remains within tens of seconds for thousands of applications, millions of client clusters, and tens of data centers.

We argue this reflects realistic platform sizes and acceptable execution time. Indeed, Krishnamurthy and Wang found roughly 400K client clusters on the Internet [19], and most infrastructure providers, such as Limelight and AT&T, operate up to 20-30 data centers. Execution time in the order of tens of seconds also seems acceptable: Oppenheimer et al., considering three real workloads, recommend application placement be done in the order of every 30 min. [24]; Wendell et al. found demand to be fairly stable on the 10-min. time scale [36].

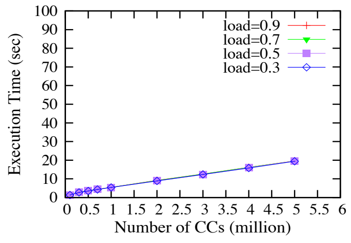
7 Prototype

To demonstrate the operation of our system, we implemented our approach and deployed a testbed that mimics a global platform. We use five machines to emulate five data centers: one in Japan, one in UK, one in Australia, one in California and one in New York. We use another five machines to mimic the clients at these locations. To emulate global deployment, we hard-code the distances between the machines representing clients and data centers using measured ping RTTs between PlanetLab nodes in the mimicked locations. We used MyXDNS [10], a DNS server configurable with external server selection policies, as a request router.

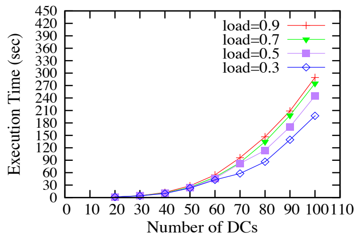
In the prototype, a decision component collects utilization and demand distribution from data centers, periodically computes placement and server selection policies using our approach, and uploads the new server selection policy into MyXDNS. MyXDNS and our decision component both run on a separate machine, updating the policy every 30 seconds. On machines that mimic data centers, we install the WebSphere application server running the TPC-W benchmark (with the browsing mix workload) as the application. We set server capacity to 100 req/s and capacity watermark at 70%. Yet we report results in terms of actual server utilization, thus justifying (at least for this application) our assumption about



(a) Execution time vs. app number



(b) Execution time vs. CC number



(c) Execution time vs. DC number

Figure 10: Scalability of Policy Computation

feasibility of using request rates as measure of demand and utilization. We use the following two scenarios to demonstrate how our system responds to the dynamically changing demand.

7.1 Demand Shift

Our first scenario shows the ability of the system to handle demand shifts from one region to another. In this scenario, we generate requests from only one location at a time and at a level that a single data center can cope with, but we change the location every 120s.

Fig. 11 shows the CPU utilization of the five machines imitating data centers. It indicates that the system handles this scenario successfully. Indeed, the application placement follows the demand after the delay induced by the periodicity of policy updates. Only one instance of the application is deployed at a time except during transitions, since our prototype is careful not to enact instance deletion until it completes pending requests - this is seen from an overlap in utilization curves.

7.2 Flash Crowd

Our second scenario imitates a flash crowd coming from one region. In this experiment, we generate requests from a single fixed location throughout the experiment but the amount of requests increases in the first 220s, then stays constant for 120s, and then drops in the final 220s. The application is initially placed in the data center in the region that generates the demand.

Fig. 12 shows the CPU utilization of the data centers in this scenario, again demonstrating successful operation of the system. Initially, data center DC_1, the nearest to client demand, is sufficient to handle the workload. As its utilization exceeds the watermark, the application is deployed at two more data centers - first at the second closest data center DC_2 and then at DC_3, the third closest. Once the flash crowd subsides, the system removes the application from the two distant data centers, first from DC_3 and then from DC_2. Note that during the flash crowd, the two closest data centers are utilized equally (up to their capacity watermark) and the more distant data center DC_3 receives only the overflow

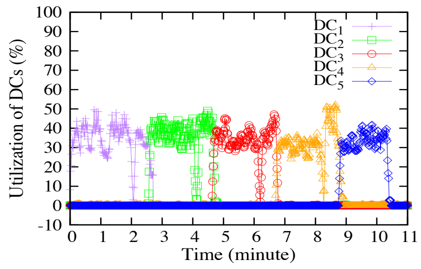


Figure 11: DC utilization with demand shift

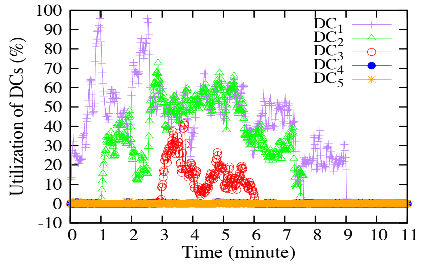


Figure 12: DC utilization with flash crowd

demand. Also note transient effects around 60 and 150 seconds, due to periodicity in policy recomputation (hence an inherent lag in reaction to changing demand) and an occasional unpredictable change in demand. E.g., at around 120 sec, the request rate produced by the demand generator unexpectedly dropped (not following the workload pattern), causing the system to lower selection probability of DC2. But right after that, the workload increased back to normal, leading to spike in utilization of DC1, while leaving DC2 only modestly utilized.

8 Related Work

While many efforts have addressed application placement and server selection, they mostly consider only one of these two problems. Schemes in [28, 22, 14, 15, 17, 18, 20] address the placement problem assuming requests are always forwarded to the closest replica. This makes these approaches suboptimal in practice as servers have limited capacity. Some works formulate global optimization problems [28, 14] but use them only as the basis for comparison since they are impractical due to computational cost. Our prefix clustering approach makes global optimization practical in many cases.

Other approaches focus on the server selection assuming a given set of replicas [11, 36, 30, 13, 31]. None of them take into account the distance between server and back-end database, partly because they mostly consider server selection for CDNs, which do not have this issue. In particular, [36] proposes an optimal decentralized server selection algorithm done by a set of mapping nodes. We address a joint placement and selection problem in a centralized manner but handle the scale issue through a novel prefix clustering technique. In [11], the authors use a min-cost flow model to generate the server selection strategy. However, they assume that the placement of applications is fixed while our approach includes the placement aspect. Furthermore, unlike our clustering technique, their approach to scalability depends on a fortuitous placement configuration.

Among the few works that tackle both placement and server selection, [25] proposes distributed placement and server selection algorithms. However, their server selection aims to balance load without considering proximity. In [29], the authors propose decentralized placement and centralized server selection algorithms that take into account both server load and proximity but compute both policies in isolation. Our unified approach showed performance advantages over it. None of them considers the distance between server and back-end database.

9 Conclusion and Future Work

This paper addresses a problem of efficient hosting of multiple applications in a globally distributed cloud computing platform and makes two main contributions. First, we design a unified approach for application placement and demand distribution policies and show its promise through both simulation experiments and a prototype testbed demonstration. Second, we propose a novel demand clustering technique and show that it makes policies based on global optimization models practical for realistic-size environments. We hope our clustering technique will be found useful beyond its application to the particular algorithms discussed here.

Important issues for future work include extending out approach to account for energy consumption, consider inter-dependencies among hosted applications, allow applications to have different priorities, and ensure pre-defined quality of service levels.

References

[1] http://download.oracle.com/docs/cd/e13222_01/wls/docs60/cluster/servlet.html.

[2] http://en.wikipedia.org/wiki/minimum_cost_flow_problem.

[3] http://mirage.cs.uoregon.edu/p2p/snapshots.html.

[4] http://www.igsystems.com/cs2/index.html.

[5] http://www.maxmind.com.

[6] http://www.mesquite.com/documentation.

[7] AHUJA, R., MAGNANTI, T., AND ORLIN, J. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[8] AKINC, U. Multi-activity facility design and location problems. *Management Science 31*, 3 (MAR 1985), pp. 275–283.

[9] AL-QUDAH, Z., ALZOUBI, H., ALLMAN, M., RABINOVICH, M., AND LIBERATORE, V. Efficient application placement in a dynamic hosting platform. In *WWW* (2009), pp. 281–290.

[10] ALZOUBI, H., RABINOVICH, M., AND SPATSCHECK, O. MyXDNS: A request routing DNS server with decoupled server selection. In *WWW* (2007), pp. 351–360.

[11] ANDREWS, M., SHEPHERD, B., SRINIVASAN, A., WINKLER, P., AND ZANE, F. Clustering and server selection using passive monitoring. In *IEEE INFOCOM* (2002), pp. 1717–1725.

[12] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ET AL. Above the clouds: A Berkeley view of cloud computing. *UC Berkeley, Tech. Rep. UCB/EECS-2009-28* (2009).

[13] BAKIRAS, S. Approximate server selection algorithms in content distribution networks. In *IEEE ICC* (2005), pp. 1490–1494.

[14] BARTOLINI, N., PRESTI, F., AND PETRIOLI, C. Optimal dynamic replica placement in Content Delivery Networks. In *IEEE ICON* (2003), pp. 125–130.

[15] CIDON, I., KUTTEN, S., AND SOFFER, R. Optimal allocation of electronic content. In *IEEE INFOCOM* (2001), pp. 205–218.

[16] GOLDBERG, A. V. An efficient implementation of a scaling minimum-cost flow algorithm. In *J. Algorithms* (1997), Academic Press, Inc.

[17] JAMIN, S., JIN, C., KURC, A., RAZ, D., AND SHAVITT, Y. Constrained mirror placement on the Internet. In *IEEE INFOCOM* (2001), pp. 31–40.

[18] JIA, X., LI, D., HU, X., AND DU, D. Placement of read-write web proxies in the internet. In *IEEE ICDCS* (2001), pp. 687–690.

[19] KRISHNAMURTHY, B., AND WANG, J. On network-aware clustering of web clients. In *ACM SIGCOMM* (2000), pp. 97–110.

[20] KRISHNAN, P., RAZ, D., AND SHAVITT, Y. The cache location problem. *IEEE/ACM ToN 8*, 5 (2000), 568–582.

[21] LAGAR-CAVILLA, H., WHITNEY, J., BRYANT, R., PATCHIN, P., BRUDNO, M., DE LARA, E., RUMBLE, S., SATYANARAYANAN, M., AND SCANNELL, A. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM TOCS 29* (2011), 2:1–2:45.

[22] LI, B., GOLIN, M., ITALIANO, G., DENG, X., AND SOHRABY, K. On the optimal placement of web proxies in the internet. In *IEEE INFOCOM* (1999), pp. 1282–1290.

[23] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26*, 6 (2004), 369–395.

[24] OPPENHEIMER, D., CHUN, B., PATTERSON, D., SNOEREN, A., AND VAHDAT, A. Service placement in a shared wide-area platform. In *USENIX ATC* (2006), pp. 26–26.

[25] PRESTI, F., PETRIOLI, C., AND VICARI, C. Distributed dynamic replica placement and request redirection in Content Delivery Networks. In *ACM/IEEE MASCOTS* (2007), pp. 366–373.

[26] QIAN, H., MILLER, E., ZHANG, W., RABINOVICH, M., AND WILLS, C. E. Agility in virtualized utility computing. In *VTDC*, pp. 9:1–9:8.

[27] QIAN, H., AND RABINOVICH, M. Application placement and demand distribution in a global cloud platform: A unified approach. Available at http://engr.case.edu/qian_hangwei/files/tech_report_global.pdf.

[28] QIU, L., PADMANABHAN, V., AND VOELKER, G. On the placement of web server replicas. In *IEEE INFOCOM* (2001), pp. 1587–1596.

[29] RABINOVICH, M., XIAO, Z., AND AGGARWAL, A. Computing on the edge: A platform for replicating internet applications. In *WCW* (2003), pp. 57–77.

[30] RANJAN, S., KARRER, R., AND KNIGHTLY, E. Wide area redirection of dynamic content by Internet data centers. In *IEEE INFOCOM* (2004), pp. 816–826.

[31] SAYAL, M., BREITBART, Y., SCHEUERMANN, P., AND VINGRALEK, R. Selection algorithms for replicated web servers. *ACM SIGMETRICS Performance Evaluation Review 26* (1998), 44–50.

[32] STEWART, C., AND SHEN, K. Performance modeling and system management for multi-component online services. In *USENIX NSDI* (2005), pp. 71–84.

[33] TANG, C., STEINDER, M., SPREITZER, M., AND PACIFICI, G. A scalable application placement controller for enterprise data centers. In *WWW* (2007), pp. 331–340.

[34] TESAURO, G., JONG, N., DAS, R., AND BENNANI, M. A hybrid reinforcement learning approach to autonomic resource allocation. In *USENIX ICAC* (2006), pp. 65–73.

[35] URGAONKAR, B., SHENOY, P., AND ROSCOE, T. Resource overbooking and application profiling in shared hosting platforms. In *USENIX OSDI* (2002), pp. 239–254.

[36] WENDELL, P., JIANG, J., FREEDMAN, M., AND REXFORD, J. DONAR: decentralized server selection for cloud services. In *ACM SIGCOMM* (2010), pp. 231–242.

[37] ZHANG, W., QIAN, H., WILLS, C., AND RABINOVICH, M. Agile resource management in a virtualized data center. In *ACM WOSP/SIPEW* (2010), pp. 129–140.

To Reserve or Not to Reserve: Optimal Online Multi-Instance Acquisition in IaaS Clouds

Wei Wang, Baochun Li, and Ben Liang
Department of Electrical and Computer Engineering
University of Toronto

Abstract—**Infrastructure-as-a-Service (IaaS) clouds offer diverse instance purchasing options. A user can either run instances on demand and pay only for what it uses, or it can prepay to reserve instances for a long period, during which a usage discount is entitled. An important problem facing a user is how these two instance options can be dynamically combined to serve time-varying demands at minimum cost. Existing strategies in the literature, however, require either exact knowledge or the distribution of demands in the long-term future, which significantly limits their use in practice. Unlike existing works, we propose two practical *online algorithms*, one deterministic and another randomized, that dynamically combine the two instance options online without any knowledge of the future. We show that the proposed deterministic (resp., randomized) algorithm incurs no more than $2 - \alpha$ (resp., $e/(e - 1 + \alpha)$) times the minimum cost obtained by an optimal *offline algorithm* that knows the exact future *a priori*, where α is the entitled discount after reservation. Our online algorithms achieve the best possible competitive ratios in both the deterministic and randomized cases, and can be easily extended to cases when short-term predictions are reliable. Simulations driven by a large volume of real-world traces show that significant cost savings can be achieved with prevalent IaaS prices.**

I. Introduction

Enterprise spending on Infrastructure-as-a-Service (IaaS) cloud is on a rapid growth path. According to [1], the public cloud services market is expected to expand from \$109 billion in 2012 to \$207 billion by 2016, during which IaaS is the fastest-growing segment with a 41.7% annual growing rate [2]. IaaS cost management therefore receives significant attention and has become a primary concern for IT enterprises.

Maintaining optimal cost management is especially challenging, given the complex pricing options offered in today's IaaS services market. IaaS cloud vendors, such as Amazon EC2, ElasticHosts, GoGrid, etc., apply diverse instance (i.e., virtual machine) pricing models at different commitment levels. At the lowest level, cloud users launch *on-demand* instances and pay only for the incurred instance-hours, without making any long-term usage commitments, e.g., [3], [4], [5]. At a higher level, there are *reserved instances* wherein users prepay a one-time upfront fee and then reserve an instance for months or years, during which the usage is either free, e.g., [4], [5], or is priced under a significant discount, e.g., [3]. Table I gives a pricing example of on-demand and reserved instances in Amazon EC2.

Acquiring instances at the cost-optimal commitment level plays a central role for cost management. Simply operating the entire load with on-demand instances can be highly inefficient.

TABLE I
PRICING OF ON-DEMAND AND RESERVED INSTANCES (LIGHT UTILIZATION, LINUX, US EAST) IN AMAZON EC2, AS OF FEB. 10, 2013.

Instance Type	Pricing Option	Upfront	Hourly
Standard Small	On-Demand	\$0	\$0.08
	1-Year Reserved	\$69	\$0.039
Standard Medium	On-Demand	\$0	\$0.16
	1-Year Reserved	\$138	\$0.078

For example, in Amazon EC2, three years of continuous on-demand service cost 3 times more than reserving instances for the same period [3]. On the other hand, naively switching to a long-term commitment incurs a huge amount of upfront payment (more than 1,000 times the on-demand rate in EC2 [3]), making reserved instances extremely expensive for sporadic workload. In particular, with time-varying loads, a user needs to answer two important questions: (1) when should I reserve instances (timing), and (2) how many instances should I reserve (quantity)?

Recently proposed instance reservation strategies, e.g., [6], [7], [8], heavily rely on long-term predictions of future demands, with historic workloads as references. These approaches, however, suffer from several significant limitations in practice. First, historic workloads might not be available, especially for startup companies who have just switched to IaaS services. In addition, not all workloads are amenable to prediction. In fact, it is observed in real production applications that workload is highly variable and statistically nonstationary [9], [10], and as a result, history may reveal very little information about the future. Moreover, due to the long span of a reservation period (e.g., 1 to 3 years in Amazon EC2), workload predictions are usually required over a very long period of time, say, years. It would be very challenging, if not impossible, to make sufficiently accurate predictions over such a long term. For all these reasons, instance reservations are usually made conservatively in practice, based on empirical experiences [11] or professional recommendations, e.g., [12], [13], [14].

In this paper, we are motivated by a practical yet fundamental question: Is it possible to reserve instances in an *online* manner, with limited or even no *a priori* knowledge of the future workload, while still incurring *near-optimal* instance acquisition costs? To our knowledge, this paper represents the first attempt to answer this question, as we make the following contributions.

With dynamic programming, we first characterize the optimal offline reservation strategy as a benchmark algorithm

(Sec. III), in which the exact future demand is assumed to be known *a priori*. We show that the optimal strategy suffers "the curse of dimensionality" [15] and is hence computationally intractable. This indicates that optimal instance reservation is in fact very difficult to obtain, even given the entire future demands.

Despite the complexity of the reservation problem in the offline setting, we present two *online* reservation algorithms, one deterministic and another randomized, that offer *the best provable* cost guarantees *without* any knowledge of future demands beforehand. We first show that our deterministic algorithm incurs no more than $2-\alpha$ times the minimum cost obtained by the benchmark optimal offline algorithm (Sec. IV), and is therefore $(2-\alpha)$*-competitive*, where $\alpha \in [0,1]$ is the entitled usage discount offered by reserved instances. This translates to a worst-case cost that is 1.51 times the optimal one under the prevalent pricing of Amazon EC2. We then establish the more encouraging result that, our randomized algorithm improves the competitive ratio to $e/(e-1+\alpha)$ in expectation, and is 1.23-competitive under Amazon EC2 pricing (Sec. V). Both algorithms achieve the *best possible* competitive ratios in the deterministic and randomized cases, respectively, and are simple enough for practical implementations. Our online algorithms can also be extended to cases when short-term predictions into the near future are reliable (Sec. VI).

In addition to our theoretical analysis, we have also evaluated both proposed online algorithms via large-scale simulations (Sec. VII), driven by Google cluster-usage traces [16] with 40 GB workload demand information of 933 users in one month. Our simulation results show that, under the pricing of Amazon EC2 [3], our algorithms closely track the demand dynamics, realizing substantial cost savings compared with several alternatives.

Though we focus on cost management of acquiring compute instances, our algorithms may find wide applications in the prevalent IaaS services market. For example, Amazon ElastiCache [17] also offers two pricing options for its web caching services, i.e., the On-Demand Cache Nodes and Reserved Cache Nodes, in which our proposed algorithms can be directly applied to lower the service costs.

II. Optimal Cost Management

We start off by briefly reviewing the pricing details of the on-demand and reservation options in IaaS clouds, based on which we formulate the online instance reservation problem for optimal cost management.

A. On-demand and Reservation Pricing

On-Demand Instances: On-demand instances let users pay for compute capacity based on usage time without long-term commitments, and are uniformly supported in leading IaaS clouds. For example, in Amazon EC2, the hourly rate of a Standard Small Instance (Linux, US East) is \$0.08 (see Table I). In this case, running it on demand for 100 hours costs a user \$8.

On-demand instances resemble the conventional pay-as-you-go model. Formally, for a certain type of instance, let the hourly rate be p. Then running it on demand for h hours incurs a cost of ph. Note that in most IaaS clouds, the hourly rate p is set as fixed in a very long time period (e.g., years), and can therefore be viewed as a constant.

Reserved Instances: Another type of pricing option that is widely supported in IaaS clouds is the reserved instance. It allows a user to reserve an instance for a long period (months or years) by prepaying an upfront reservation fee, after which, the usage is either free, e.g., ElasticHosts [4], GoGrid [5], or is priced with a heavy discount, e.g., Amazon EC2 [3]. For example, in Amazon EC2, to reserve a Standard Small Instance (Linux, US East, Light Utilization) for 1 year, a user pays an upfront \$69 and receives a discount rate of \$0.039 per hour within 1 year of the reservation time, as oppose to the regular rate of \$0.08 (see Table I). Suppose this instance has run 100 hours before the reservation expires. Then the total cost incurred is \$69 + 0.039×100 = \$72.9.

Reserved instances resemble the wholesale market. Formally, for a certain type of reserved instance, let the reservation period be τ (counted by the number of hours). An instance that is reserved at hour i would expire before hour $i+\tau$. Without loss of generality, we assume the reservation fee to be 1 and normalize the on-demand rate p to the reservation fee. Let $\alpha \in [0,1]$ be the received discount due to reservation. A reserved instance running for h hours during the reservation period incurs a discounted running cost αph plus a reservation fee, leading to a total cost of $1+\alpha ph$. In the previous example, the normalized on-demand rate $p = 0.08/69$; the received discount due to reservation is $\alpha = 0.039/0.08 = 0.49$; the running hour $h = 100$; and the normalized overall cost is

$$1 + \alpha ph = 72.9/69 \ .$$

In practice, cloud providers may offer multiple types of reserved instances with different reservation periods and utilization levels. For example, Amazon EC2 offers 1-year and 3-year reservations with light, medium, and high utilizations [3]. For simplicity, we limit the discussion to one type of such reserved instances chosen by a user based on its rough estimations. We also assume that the on-demand rate is far smaller than the reservation fee, i.e., $p \ll 1$, which is always the case in IaaS clouds, e.g., [3], [4], [5].

B. The Online Instance Reservation Problem

In general, launching instances on demand is more cost efficient for sporadic workload, while reserved instances are more suitable to serve stable demand lasting for a long period of time, for which the low hourly rate would compensate for the high upfront fee. The cost management problem is to optimally combine the two instance options to serve the time-varying demand, such that the incurred cost is minimized. In this section, we consider making instance purchase decisions *online*, without any *a priori* knowledge about the future demands. Such an online model is especially important for startup companies who have limited or no history demand data and those cloud users whose workloads are highly variable and non-stationary — in both cases reliable predictions are unavailable. We postpone the discussions for cases when short-term demand predictions are reliable in Sec. VI.

Since IaaS instances are billed in an hourly manner, we slot the time to a sequence of hours indexed by $t = 0, 1, 2, \ldots$ At each time t, demand d_t arrives, meaning that a user requests d_t instances, $d_t = 0, 1, 2, \ldots$ To accommodate this demand, the user decides to use o_t on-demand instances and $d_t - o_t$ reserved instances. If the previously reserved instances that remain available at time t are fewer than $d_t - o_t$, then new instances need to be reserved. Let r_t be the number of instances that are *newly reserved* at time t, $r_t = 0, 1, 2, \ldots$ The overall cost incurred at time t is the on-demand cost $o_t p$ plus the reservation cost $r_t + \alpha p(d_t - o_t)$, where r_t is the upfront payments due to new reservations, and $\alpha p(d_t - o_t)$ is the cost of running $d_t - o_t$ reserved instances.

The cost management problem is to make instance purchase decisions online, i.e., r_t and o_t at each time t, before seeing future demands $d_{t+1}, d_{t+2}, \ldots$ The objective is to minimize the overall instance acquisition costs. Suppose demands last for an arbitrary time T (counted by the number of hours). We have the following *online instance reservation* problem:

$$\begin{aligned}
\min_{\{r_t, o_t\}} \quad & C = \sum_{t=1}^{T} (o_t p + r_t + \alpha p(d_t - o_t)) \ , \\
\text{s.t.} \quad & o_t + \sum_{i=t-\tau+1}^{t} r_i \geq d_t \ , \\
& o_t, r_t \in \{0, 1, 2, \ldots\}, t = 1, \ldots, T \ .
\end{aligned} \tag{1}$$

Here, the first constraint ensures that all d_t instances demanded at time t are accommodated, with o_t on-demand instances and $\sum_{i=t-\tau+1}^{t} r_i$ reserved instances that remain active at time t. Note that instances that are reserved before time $t - \tau + 1$ have all expired at time t, where τ is the reservation period. For convenience, we set $r_t = 0$ for all $t \leq 0$.

The main challenge of problem (1) lies in its online setting. Without knowledge of future demands, the online strategy may make purchase decisions that turn out later not to be optimal. Below we clarify the performance metrics to measure how far away an online strategy may deviate from the optimal solution.

C. Measure of Competitiveness

To measure the cost performance of an online strategy, we adopt the standard *competitive analysis* [18]. The idea is to bound the gap between the cost of an interested online algorithm and that of the optimal offline strategy. The latter is obtained by solving problem (1) with the exact future demands $d_1, \ldots, d_T$ given *a priori*. Formally, we have

Definition 1 (Competitive analysis): A *deterministic* online reservation algorithm A is *c-competitive* (c is a constant) if for all possible demand sequences $\mathbf{d} = \{d_1, \ldots, d_T\}$, we have

$$C_A(\mathbf{d}) \leq c \cdot C_{\mathrm{OPT}}(\mathbf{d}) \ , \tag{2}$$

where $C_A(\mathbf{d})$ is the instance acquisition cost incurred by algorithm A given input $\mathbf{d}$, and $C_{\mathrm{OPT}}(\mathbf{d})$ is the optimal instance acquisition cost given input $\mathbf{d}$. Here, $C_{\mathrm{OPT}}(\mathbf{d})$ is obtained by solving the instance reservation problem (1) *offline*, where the exact demand sequence $\mathbf{d}$ is assumed to know *a priori*.

A similar definition of the competitive analysis also extends to the *randomized* online algorithm A, where the decision making is drawn from a random distribution. In this case, the LHS of (2) is simply replaced by $\mathbf{E}[C_A(\mathbf{d})]$, the expected cost of randomized algorithm A given input $\mathbf{d}$. (See [18] for a detailed discussion.)

Competitive analysis takes an optimal offline algorithm as a benchmark to measure the cost performance of an online strategy. Intuitively, the smaller the competitive ratio c is, the more closely the online algorithm A approaches the optimal solution. Our objective is to design *optimal online algorithms* with the smallest competitive ratio.

We note that the instance reservation problem (1) captures the Bahncard problem [19] as a special case when a user demands no more than one instance at a time, i.e., $d_t \leq 1$ for all t. The Bahncard problem models online ticket purchasing on the German Federal Railway, where one can opt to buy a Bahncard (reserve an instance) and to receive a discount on all trips within one year of the purchase date. It has been shown in [19], [20] that the lower bound of the competitive ratio is $2 - \alpha$ and $e/(e - 1 + \alpha)$ for the deterministic and randomized Bahncard algorithms, respectively. Because the Bahncard problem is a special case of our problem (1), we have

Lemma 1: The competitive ratio of problem (1) is *at least* $2 - \alpha$ for deterministic online algorithms, and is at least $e/(e - 1 + \alpha)$ for randomized online algorithms.

However, we show in the following that the instance reserving problem (1) is by no means a trivial extension to the Bahncard problem, mainly due to the time-multiplexing nature of reserved instances.

D. Bahncard Extension and Its Inefficiency

A natural way to extend the Bahncard solutions in [19] is to decompose problem (1) into separate Bahncard problems. To do this, we introduce a set of *virtual users* indexed by 1, 2, ... Whenever demand d_t arises at time t, we view the original user as d_t virtual users 1, 2, ..., d_t, each requiring one instance at that time. Each virtual user then reserves instances (i.e., buy a Bahncard) separately to minimize its cost, which is exactly a Bahncard problem.

However, such an extension is highly inefficient. An instance reserved by one virtual user, even idle, can *never* be multiplexed with another, who still needs to pay for its own demand. For a real user, this implies that it has to acquire additional instances, either on-demand or reserved, even if the user has already reserved sufficient amount of instances to serve its demand, which inevitably incurs a large amount of unnecessary cost.

We learn from the above failure that instances must be reserved *jointly* and *time multiplexed* appropriately. These factors significantly complicate our problem (1). Indeed, as we see in the next section, even with full knowledge of the future demand, obtaining an optimal offline solution to (1) is computationally prohibitive.

III. The Offline Strategy and Its Intractability

In this section we consider the benchmark *offline cost management* strategy for problem (1), in which the exact future demands are given *a priori*. The offline setting is an integer programming problem and is generally difficult to solve. We derive the optimal solution via dynamic programming. However, such an optimal offline strategy suffers from "the curse of dimensionality" [15] and is computationally intractable.

We start by defining states. A state at time t is defined as a $(\tau-1)$-tuple $\mathbf{s}_t = (s_{t,1}, \dots, s_{t,\tau-1})$, where $s_{t,i}$ denotes the number of instances that are reserved *no later than* t and *remain active* at time $t+i$, $i = 1, \dots, \tau-1$. We use a $(\tau-1)$-tuple to define a state because an instance that is reserved no later than t will no longer be active at time $t+\tau$ and thereafter. Clearly, $s_{t,1} \geq \cdots \geq s_{t,\tau-1}$ as reservations gradually expire.

We make an important observation, that state $\mathbf{s}_t$ only depends on states $\mathbf{s}_{t-1}$ at the previous time, and is *independent* of earlier states $\mathbf{s}_{t-2}, \dots, \mathbf{s}_1$. Specifically, suppose state $\mathbf{s}_{t-1}$ is reached at time $t-1$. At the beginning of the next time t, r_t new instances are reserved. These newly reserved r_t instances will add to the active reservations starting from time t, leading state $\mathbf{s}_{t-1}$ to transit to $\mathbf{s}_t$ following the transition equations below:

$$\begin{cases} s_{t,i} = s_{t-1,i+1} + r_t, & i = 1, \dots, \tau-2\ ; \\ s_{t,\tau-1} = r_t. \end{cases} \tag{3}$$

Let $V(\mathbf{s}_t)$ be the minimum cost of serving demands $d_1, \dots, d_t$ up to time t, conditioned upon the fact that state $\mathbf{s}_t$ is reached at time t. We have the following recursive Bellman equations:

$$V(\mathbf{s}_t) = \min_{\mathbf{s}_{t-1}} \{V(\mathbf{s}_{t-1}) + c(\mathbf{s}_{t-1}, \mathbf{s}_t)\}, \quad t > 0, \tag{4}$$

where $c(\mathbf{s}_{t-1}, \mathbf{s}_t)$ is the transition cost, and the minimization is over all states $\mathbf{s}_{t-1}$ that can transit to $\mathbf{s}_t$ following the transition equations (3). The Bellman equations (4) indicate that the minimum cost of reaching $\mathbf{s}_t$ is given by the minimum cost of reaching a previous state $\mathbf{s}_{t-1}$ plus the transition cost $c(\mathbf{s}_{t-1}, \mathbf{s}_t)$, minimized over all possible previous states $\mathbf{s}_{t-1}$. Let

$$X^+ = \max\{0, X\}\ . \tag{5}$$

The transition cost is defined as

$$c(\mathbf{s}_{t-1}, \mathbf{s}_t) = o_t p + r_t + \alpha p(d_t - o_t)\ , \tag{6}$$

where

$$r_t = s_{t,\tau-1}, \tag{7}$$

$$o_t = (d_t - r_t - s_{t-1,1})^+, \tag{8}$$

and the transition from $\mathbf{s}_{t-1}$ to $\mathbf{s}_t$ follows (3). The rationale of (6) is straightforward. By the transition equations (3), state $\mathbf{s}_{t-1}$ transits to $\mathbf{s}_t$ by reserving $r_t = s_{t,\tau-1}$ instances at time t. Adding the $s_{t-1,1}$ instances that have been reserved before t, we have $r_t + s_{t-1,1}$ reserved instances to use at time t. We therefore need $o_t = (d_t - r_t - s_{t-1,1})^+$ on-demand instances at that time.

The boundary conditions of Bellman equations (4) are

$$V(\mathbf{s}_0) = s_{0,1}, \quad \text{for all } \mathbf{s}_0 = (s_{0,1}, \dots, s_{0,\tau-1}), \tag{9}$$

because an initial state $\mathbf{s}_0$ indicates that a user has already reserved $s_{0,1}$ instances at the beginning and paid $s_{0,1}$.

With the analyses above, we see that the dynamic programming defined by (3), (4), (6), and (9) optimally solves the offline instance reserving problem (1). Therefore, it gives $C_{\text{OPT}}(\mathbf{d})$ in theory.

Unfortunately, the dynamic programming presented above is *computationally intractable*. This is because to solve the Bellman equations (4), one has to compute $V(\mathbf{s}_t)$ for all states $\mathbf{s}_t$. However, since a state $\mathbf{s}_t$ is defined in a high-dimensional space — recall that $\mathbf{s}_t$ is defined as a $(\tau-1)$-tuple — there exist *exponentially* many such states. Therefore, looping over all of them results in exponential time complexity. This is known as the curse of dimensionality suffered by high-dimensional dynamic programming [15].

The intractability of the offline instance reservation problem (1) suggests that optimal cost management in IaaS clouds is in fact a very complicate problem, even if future demands can be accurately predicted. However, we show in the following sections that it is possible to have online strategies that are highly efficient with near-optimal cost performance, even without any knowledge of the future demands.

IV. Optimal Deterministic Online Strategy

In this section, we present a deterministic online reservation strategy that incurs no more than $2-\alpha$ times the minimum cost. As indicated by Lemma 1, this is also the best that one can expect from a deterministic algorithm.

A. The Deterministic Online Algorithm

We start off by defining a *break-even point* at which a user is indifferent between using a reserved instance and an on-demand instance. Suppose an on-demand instance is used to accommodate workload in a time interval that spans a reservation period, incurring a cost c. If we use a reserved instance instead to serve the same demand, the cost will be $1+\alpha c$. When $c = 1/(1-\alpha)$, both instances cost the same, and are therefore indifferent to the user. We hence define the break-even point as

$$\beta = 1/(1-\alpha)\ . \tag{10}$$

Clearly, the use of an on-demand instance is well justified *if and only if* the incurred cost does not exceed the break-even point, i.e., $c \leq \beta$.

Our deterministic online algorithm is summarized as follows. By default, all workloads are assumed to be operated with on-demand instances. At time t, upon the arrival of demand d_t, we check the use of on-demand instances in a recent reservation period, starting from time $t-\tau+1$

to t, and reserve a new instance whenever we see an on-demand instance incurring more costs than the break-even point. Algorithm 1 presents the detail.

Algorithm 1 Deterministic Online Algorithm A_β

```
1. Let $x_i$ be the number of reserved instances at time $i$,
   Initially, $x_i \leftarrow 0$ for all $i = 0, 1, \dots$
2. Let $I(X)$ be an indicator function where $I(X) = 1$ if $X$ is
   true and $I(X) = 0$ otherwise. Also let $X^+ = \max\{X, 0\}$.
3. Upon the arrival of demand $d_t$, loop as follows:
4. while $p \sum_{i=t-\tau+1}^{t} I(d_i > x_i) > \beta$ do
5.     Reserve a new instance: $r_t \leftarrow r_t + 1$.
6.     Update the number of reservations that can be used in
       the future: $x_i \leftarrow x_i + 1$ for $i = t, \dots, t + \tau - 1$.
7.     Add a "phantom" reservation to the recent period,
       indicating that the history has already been "processed":
       $x_i \leftarrow x_i + 1$ for $i = t - \tau + 1, \dots, t - 1$.
8. end while
9. Launch on-demand instances: $o_t \leftarrow (d_t - x_t)^+$.
10. $t \leftarrow t + 1$, repeat from 3.
```

Fig. 1 helps to illustrate Algorithm 1. Whenever demand d_t arises, we check the recent reservation period from time $t - \tau + 1$ to t. We see that an on-demand instance has been used at time i if demand d_i exceeds the number of reservations x_i (both actual and phantom), $i = t-\tau+1, \dots, t$. The shaded area in Fig. 1 represents the use of an on-demand instance in the recent period, which incurs a cost of $p \sum_{i=t-\tau+1}^{t} I(d_i > x_i)$. If this cost exceeds the break-even point β (line 4 of Algorithm 1), then such use of an on-demand instance is not well justified: We *should have* reserved an instance before at time $t-\tau+1$ and used it to serve the demand (shaded area) instead, which *would have* lowered the cost. As a compensation for this "mistake," we reserve an instance at the current time t (line 5), and will have one more reservation to use in the future (line 6). Since we have already compensated for a misuse of an on-demand instance (the shaded area), we add a "phantom" reservation to the history so that such a mistake will not be counted multiple times in the following rounds (line 7). This leads to an update of the reservation number $\{x_i\}$ (see the bottom figure in Fig. 1).

Unlike the simple extension of the Bahncard algorithm described in Sec. II-D, Algorithm 1 jointly reserves instances by taking both the currently active reservations (i.e., x_t) and the historic records (i.e., x_i, $i < t$) into consideration (line 4), without any knowledge of the future. We will see later in Sec. VII that such a joint reservation significantly outperforms the Bahncard extension where instances are reserved separately.

B. Performance Analysis: $(2 - \alpha)$-Competitiveness

The "trick" of Algorithm 1 is to make reservations "lazily": no instance is reserved unless the misuse of an on-demand instance is seen. Such a "lazy behaviour" turns out to guarantee that the algorithm incurs no more than $2 - \alpha$ times the minimum cost.

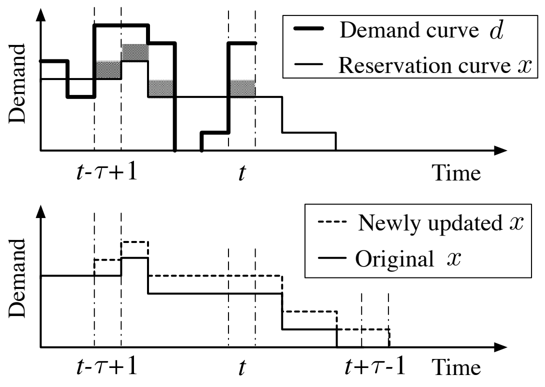


Fig. 1. Illustration of Algorithm 1. The shaded area in the top figure shows the use of an on-demand instance in the recent period. An instance is reserved at time t if the use of this on-demand instance is not well justified. The bottom figure shows the corresponding updates of the reservation curve x.

Let A_β denote Algorithm 1 and let OPT denote the optimal offline algorithm. We now make an important observation, that OPT reserves at least the same amount of instances as A_β does, for any demand sequence.

Lemma 2: Given an arbitrary demand sequence, let n_β be the number of instances reserved by A_β, and let n_{OPT} be the number of instances reserved by OPT. Then $n_\beta \le n_{\text{OPT}}$.

Lemma 2 can be viewed as a result of the "lazy behaviour" of A_β, in which instances are reserved just to compensate for the previous "purchase mistakes." Intuitively, such a conservative reservation strategy leads to fewer reserved instances. The proof of Lemma 2, however, is tedious and is deferred to our technical report [21].

We are now ready to analyze the cost performance of A_β, using the optimal offline algorithm OPT as a benchmark.

Proposition 1: Algorithm 1 is $(2 - \alpha)$-*competitive*. Formally, for any demand sequence,

$$C_{A_\beta} \le (2 - \alpha) C_{\text{OPT}} , \qquad (11)$$

where C_{A_β} is the cost of Algorithm 1 (A_β), and C_{OPT} is the cost of the optimal offline algorithm OPT.

Proof: Suppose A_β (resp., OPT) launches o_t (resp., o_t^*) on-demand instances at time t. Let $\text{Od}(A_\beta)$ be the costs incurred by these on-demand instances under A_β, i.e., $\text{Od}(A_\beta) = \sum_{t=1}^{T} o_t p$. We refer to $\text{Od}(A_\beta)$ as the *on-demand costs* of A_β. Similarly, we define the on-demand costs incurred by OPT as $\text{Od}(\text{OPT}) = \sum_{t=1}^{T} o_t^* p$. Also, let $\text{Od}(A_\beta \backslash \text{OPT}) = \sum_{t=1}^{T} (o_t - o_t^*)^+ p$ be the on-demand costs incurred in A_β that are not incurred in OPT. We see

$$\text{Od}(A_\beta \backslash \text{OPT}) \le \beta n_{\text{OPT}} \qquad (12)$$

by noting the following two facts: First, demands $\sum_{t=1}^{T} (o_t - o_t^*)^+$ are served by at most n_{OPT} reserved instances in OPT. Second, demands that are served by the same reserved instance in OPT incur on-demand costs of at most β in A_β (by the definition of A_β). We therefore bound $\text{Od}(A_\beta)$ as follows:

$$\begin{aligned} \text{Od}(A_\beta) &\le \text{Od}(\text{OPT}) + \text{Od}(A_\beta \backslash \text{OPT}) \\ &\le \text{Od}(\text{OPT}) + \beta n_{\text{OPT}} . \end{aligned} \qquad (13)$$

Let $S = \sum_{t=1}^{T} d_t p$ be the cost of serving all demands with on-demand instances. We bound the cost of OPT as follows:

$$\begin{aligned} C_{\mathrm{OPT}} &= \mathrm{Od}(\mathrm{OPT}) + n_{\mathrm{OPT}} + \alpha(S - \mathrm{Od}(\mathrm{OPT})) && (14)\\ &\geq \mathrm{Od}(\mathrm{OPT}) + n_{\mathrm{OPT}} + \alpha\beta n_{\mathrm{OPT}} && (15)\\ &\geq n_{\mathrm{OPT}}/(1-\alpha) \ . && (16)\end{aligned}$$

Here, (15) holds because in OPT, demands that are served by the same reserved instance incur at least a break-even cost β when priced at an on-demand rate p.

With (13) and (16), we bound the cost of A_β as follows:

$$\begin{aligned} C_{A_\beta} &= \mathrm{Od}(A_\beta) + n_\beta + \alpha(S - \mathrm{Od}(A_\beta)) \\ &\leq (1-\alpha)\mathrm{Od}(A_\beta) + n_{\mathrm{OPT}} + \alpha S && (17)\\ &\leq (1-\alpha)(\mathrm{Od}(\mathrm{OPT}) + \beta n_{\mathrm{OPT}}) + \alpha S + n_{\mathrm{OPT}} && (18)\\ &= C_{\mathrm{OPT}} + n_{\mathrm{OPT}} && (19)\\ &\leq (2-\alpha) C_{\mathrm{OPT}} \ . && (20)\end{aligned}$$

Here, (17) holds because $n_\beta \leq n_{\mathrm{OPT}}$ (Lemma 2). Inequality (18) follows from (13), while (20) is derived from (16). ■

By Lemma 1, we see that $2-\alpha$ is already the best possible competitive ratio for deterministic online algorithms, which implies that Algorithm 1 is optimal in a view of competitive analysis.

Proposition 2: Among all online deterministic algorithms of problem (1), Algorithm 1 is *optimal* with the smallest competitive ratio of $2-\alpha$.

As a direct application, in Amazon EC2 with reservation discount $\alpha = 0.49$ (see Table I), algorithm A_β will lead to no more than 1.51 times the optimal instance purchase cost.

Despite the already satisfactory cost performance offered by the proposed deterministic algorithm, we show in the next section that the competitive ratio may be further improved if randomness is introduced.

V. Optimal Randomized Online Strategy

In this section, we construct a randomized online strategy that is a random distribution over a family of deterministic online algorithms similar to A_β. We show that such randomization improves the competitive ratio to $e/(e-1+\alpha)$ and hence leads to a better cost performance. As indicated by Lemma 1, this is the best that one can expect without knowledge of future demands.

We start by defining a family of algorithms similar to the deterministic algorithm A_β. Let A_z be a similar deterministic algorithm to A_β with β in line 4 of Algorithm 1 replaced by $z \in [0, \beta]$. That is, A_z reserves an instance whenever it sees an on-demand instance incurring more costs than z in the recent reservation period. Intuitively, the value of z reflects the *aggressiveness* of a reservation strategy. The smaller the z, the more aggressive the strategy. As an extreme, a user will always reserve when $z = 0$. Another extreme goes to $z = \beta$ (Algorithm 1), in which the user is very conservative in reserving new instances.

Our randomized online algorithm picks a $z \in [0, \beta]$ according to a density function $f(z)$ and runs the resulting algorithm A_z. Specifically, the density function $f(z)$ is defined as

$$f(z) = \begin{cases} (1-\alpha)e^{(1-\alpha)z}/(e-1+\alpha), & z \in [0,\beta), \\ \delta(z-\beta)\cdot\alpha/(e-1+\alpha), & \text{o.w.}, \end{cases} \qquad (21)$$

where $\delta(\cdot)$ is the *Dirac delta function*. That is, we pick $z = \beta$ with probability $\alpha/(e-1+\alpha)$. It is interesting to point out that in other online rent-or-buy problems, e.g., [22], [20], [23], the density function of a randomized algorithm is usually continuous[1]. However, we note that a continuous density function does not lead to the minimum competitive ratio in our problem. Algorithm 2 formalizes the descriptions above.

Algorithm 2 Randomized Online Algorithm

```
1. Randomly pick $z \in [0, \beta]$ according to a density function
   $f(z)$ defined by (21)
2. Run $A_z$
```

The rationale behind Algorithm 2 is to strike a suitable balance between reserving "aggressively" and "conservatively." Intuitively, being aggressive is cost efficient when future demands are long-lasting and stable, while being conservative is efficient for sporadic demands. Given the unknown future, the algorithm randomly chooses a strategy A_z, with an expectation that the incurred cost will closely approach the *ex post* minimum cost. The following theorem shows that the choice of $f(z)$ in (21) leads to the optimal competitive ratio $e/(e-1+\alpha)$. The proof is given in [21].

Proposition 3: Algorithm 2 is $e/(e-1+\alpha)$-competitive. Formally, for any demand sequence,

$$\mathbf{E}[C_{A_z}] \leq \frac{e}{e-1+\alpha} C_{\mathrm{OPT}} \ , \qquad (22)$$

where the expectation is over z between 0 and β according to density function $f(z)$ defined in (21).

By Lemma 1, we see that no online randomized algorithm is better than Algorithm 2 in terms of the competitive ratio.

Proposition 4: Among all online randomized algorithms of problem (1), Algorithm 2 is optimal with the smallest competitive ratio $e/(e-1+\alpha)$.

As a direct application, in Amazon EC2 with reservation discount $\alpha = 0.49$ (see Table I), the randomized algorithm will lead to a competitive ratio of 1.23, compared with the 1.51-competitiveness of the deterministic alternative.

VI. Cost Management with Short-Term Demand Predictions

In the previous sections, our discussions focus on the extreme cases, with either full future demand information (i.e., the offline case in Sec. III) or no *a priori* knowledge of the future (i.e., the online case in Sec. IV and V). In this section, we consider the middle ground in which short-term demand predictions are reliable. For example, websites typically see diurnal patterns exhibited on their workloads, based on which

[1]The density function in these works is chosen as $f(z) = e^z/(e-1), z \in [0,1]$, which is a special case of ours when $\alpha = 0$.

it is possible to have a demand prediction window that is weeks into the future. Both our online algorithms can be easily extended to utilize these knowledge of future demands when making reservation decisions.

We begin by formulating the instance reservation problem with limited information of future demands. Let w be the prediction window. That is, at any time t, a user can predict its future demands $d_{t+1}, \ldots, d_{t+w}$ in the next w hours. Since only short-term predictions are reliable, one can safely assume that the prediction window is less than a reservation period, i.e., $w < \tau$. The instance reservation problem resembles the online reservation problem (1), except that the instance purchase decisions made at each time t, i.e., the number of reserved instances (r_t) and on-demand instances (o_t), are based on both history and future demands predicted, i.e., $d_1, \ldots, d_{t+w}$. The competitive analysis (Definition 1) remains valid in this case.

The Deterministic Algorithm: We extend our deterministic online algorithm as follows. As before, all workloads are *by default* served by on-demand instances. At time t, we can predict the demands up to time $t+w$. Unlike the online deterministic algorithm, we check the use of on-demand instances in a reservation period across *both history and future*, starting from time $t+w-\tau+1$ to $t+w$. A new instance is reserved at time t whenever we see an on-demand instance incurring more costs than the break-even point β and the currently effective reservations are less than the current demand d_t. Algorithm 3, also denoted by A_β^w, shows the details.

Algorithm 3 Deterministic Algorithm A_β^w with Prediction Window w

```
1. Let $x_i$ be the number of reserved instances at time $i$,
   Initially, $x_i \leftarrow 0$ for all $i = 0, 1, \ldots$
2. Upon the arrival of demand $d_t$, loop as follows:
3. while $p \sum_{i=t+w-\tau+1}^{t+w} I(d_i > x_i) > \beta$ and $x_t < d_t$ do
4.     Reserve a new instance: $r_t \leftarrow r_t + 1$.
5.     Update the number of reservations that can be used in
       the future: $x_i \leftarrow x_i + 1$ for $i = t, \ldots, t+\tau-1$.
6.     Add a "phantom" reservation to the history, indicating
       that the history has already been "processed": $x_i \leftarrow$
       $x_i + 1$ for $i = t+w-\tau+1, \ldots, t-1$.
7. end while
8. Launch on-demand instances: $o_t \leftarrow (d_t - x_t)^+$.
9. $t \leftarrow t+1$, repeat from 2.
```

The Randomized Algorithm: The randomized algorithm can also be constructed as a random distribution over a family of deterministic algorithms similar to A_β^w. In particular, let A_z^w be similarly defined as algorithm A_β^w with β replaced by $z \in [0, \beta]$ in line 3 of Algorithm 3. The value of z reflects the aggressiveness of instance reservation. The smaller the z, the more aggressive the reservation strategy. Similar to the online randomized, we introduce randomness to strike a good balance between reserving aggressively and conservatively. Our algorithm randomly picks $z \in [0, \beta]$ according to the same density function $f(z)$ defined by (21), and runs the resulting algorithm A_z^w. Algorithm 4 formalizes the description above.

Algorithm 4 Randomized Algorithm with Prediction Window w

```
1. Randomly pick $z \in [0, \beta]$ according to a density function
   $f(z)$ defined by (21)
2. Run $A_z^w$
```

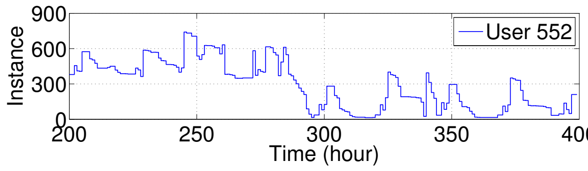


Fig. 2. The demand curve of User 552 in Google cluster-usage traces [16], over 1 month.

It is easy to see that both the deterministic and the randomized algorithms presented above improve the cost performance of their online counterparts, due to the knowledge of future demands. Therefore, we have Proposition 5 below. We will quantify their performance gains via trace-driven simulations in the next section.

Proposition 5: Algorithm 3 is $(2-\alpha)$-competitive, and Algorithm 4 is $e/(e-1+\alpha)$-competitive.

VII. Trace-Driven Simulations

So far, we have analyzed the cost performance of the proposed algorithms in a view of competitive analysis. In this section, we evaluate their performance for practical cloud users via simulations driven by a large volume of real-world traces.

A. Dataset Description and Preprocessing

Long-term user demand data in public IaaS clouds are often confidential: no cloud provider has released such information so far. For this reason, we turn to Google cluster-usage traces that were recently released in [16]. Although Google is not a public IaaS cloud, its cluster-usage traces record the computing demands of its cloud services and Google engineers, which can represent the computing demands of IaaS users to some degree. The dataset contains 40 GB of workload resource requirements (e.g., CPU, memory, disk, etc.) of 933 users over 29 days in May 2011, on a cluster of more than 11K Google machines.

Demand Curve: Given the workload traces of each user, we ask the question: How many computing instances would this user require if it were to run the same workload in a public IaaS cloud? For simplicity, we set an instance to have the same computing capacity as a cluster machine, which enables us to accurately estimate the run time of computational tasks by learning from the original traces. We then schedule these tasks onto instances with sufficient resources to accommodate their requirements. Computational tasks that cannot run on the same server in the traces (e.g., tasks of MapReduce) are scheduled to different instances. In the end, we obtain a demand curve for each user, indicating how many instances this user requires in each hour. Fig. 2 illustrates such a demand curve for a user.

User Classification: To investigate how our online algorithms perform under different demand patterns, we classify

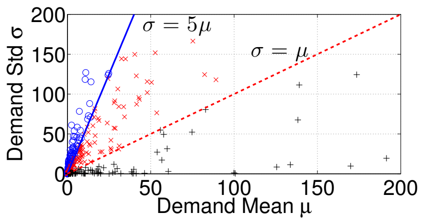


Fig. 3. User demand statistics and group division.

all 933 users into three groups by the *demand fluctuation level* measured as the ratio between the standard deviation σ and the mean μ.

Specifically, *Group 1* consists of users whose demands are highly fluctuating, with $\sigma/\mu \geq 5$. As shown in Fig. 3 (circle 'o'), these demands usually have small means, which implies that they are highly sporadic and are best served with on-demand instances. *Group 2* includes users whose demands are less fluctuating, with $1 \leq \sigma/\mu < 5$. As shown in Fig. 3 (cross 'x'), these demands cannot be simply served by on-demand or reserved instances alone. *Group 3* includes all remaining users with relatively stable demands ($0 \leq \sigma/\mu < 1$). As shown in Fig. 3 (plus '+'), these demands have large means and are best served with reserved instances. Our evaluations are carried out for each user group.

Pricing: Throughout the simulation, we adopt the pricing of Amazon EC2 standard small instances with the on-demand rate \$0.08, the reservation fee \$69, and the discount rate \$0.039 (Linux, US East, 1-year light utilization). Since the Google traces only span one month, we proportionally shorten the on-demand billing cycle from one hour to one minute, and the reservation period from 1 year to 6 days (i.e., $24 \times 365 = 8760$ minutes $= 6$ days) as well.

B. Evaluations of Online Algorithms

We start by evaluating the performance of online algorithms without any *a priori* knowledge of user demands.

Benchmark Online Algorithms: We compare our online deterministic and randomized algorithms with three benchmark online strategies. The first is *All-on-demand*, in which a user never reserves and operates all workloads with on-demand instances. This algorithm, though simple, is the most common strategy in practice, especially for those users with time-varying workloads [11]. The second algorithm is *All-reserved*, in which all computational demands are served via reservations. The third online algorithm is the simple extension to the Bahncard algorithm proposed in [19] (see Sec. II-D), and is referred to as *Separate* because instances are reserved separately. All three benchmark algorithms, as well as the two proposed online algorithms, are carried out for each user in the Google traces. All the incurred costs are ***normalized to All-on-demand***.

Cost Performance: We present the simulation results in Fig. 4, where the CDF of the normalized costs are given, grouped by users with different demand fluctuation levels. We see in Fig. 4a that when applied to all 933 users, both

TABLE II
AVERAGE COST PERFORMANCE (NORMALIZED TO ALL-ON-DEMAND).

Algorithm	**All users**	**Group 1**	**Group 2**	**Group 3**
All-reserved	16.48	48.99	1.25	0.61
Separate	0.88	1.01	1.02	0.71
Deterministic	0.81	1.00	0.89	0.67
Randomized	0.76	1.02	0.79	0.63

the deterministic and randomized online algorithms realize significant cost savings compared with all three benchmarks. In particular, when switching from All-on-demand to the proposed online algorithms, more than 60% users cut their costs. About 50% users save more than 40%. Only 2% incur slightly more costs than before. For users who switch from All-reserved to our randomized online algorithms, the improvement is even more substantial. As shown in Fig. 4a, cost savings are almost guaranteed, with 30% users saving more than 50%. We also note that Separate, though generally outperforms All-on-demand and All-reserved, incurs more costs than our online algorithms, mainly due to its ignorance of reservation correlations.

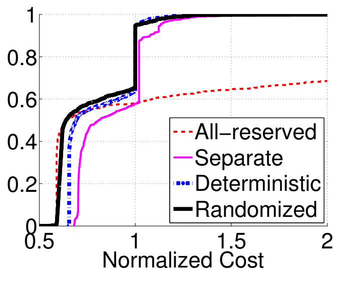
We next compare the cost performance of all five algorithms at different demand fluctuation levels. As expected, when it comes to the extreme cases, All-on-demand is the best fit for Group 1 users whose demands are known to be highly busty and sporadic (Fig. 4b), while All-reserved incurs the least cost for Group 3 users with stable workloads (Fig. 4d). These two groups of users, should they know their demand patterns, would have the least incentive to adopt advanced instance reserving strategies, as naively switching to one option is already optimal. However, even in these extreme cases, our online algorithms, especially the randomized one, remain highly competitive, incurring only slightly higher cost.

However, the acquisition of instances is not always a black-and-white choice between All-on-demand and All-reserved. As we observe from Fig. 4c, for Group 2 users, a more intelligent reservation strategy is essential, since naive algorithms, either All-on-demand or All-reserved, are always highly risky and can easily result in skyrocketing cost. Our online algorithms, on the other hand, become the best choices in this case, outperforming all three benchmark algorithms by a significant margin.

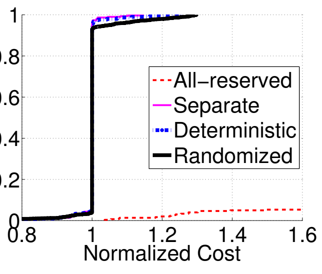
Table II summarizes the average cost performance for each user group. We see that, in all cases, our online algorithms remain highly competitive, incurring near-optimal costs for a user.

C. The Value of Short-Term Predictions

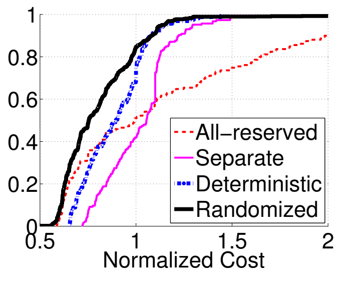
While our online algorithms perform sufficiently well without knowledge of future demands, we show in this section that more cost savings are realized by their extensions when short-term demand predictions are reliable. In particular, we consider three prediction windows that are 1, 2, and 3 months into the future, respectively. For each prediction window, we run both the deterministic and randomized extensions (i.e., Algorithm 3 and 4) for each Google user in the traces, and compare their costs with those incurred by the online counterparts without



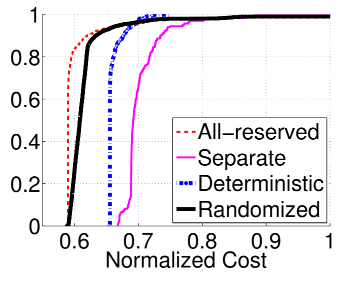
(a) Cost CDF (all users)



(b) Cost CDF (high fluctuation)

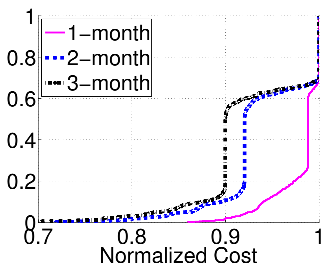


(c) Cost CDF (medium fluctuation)

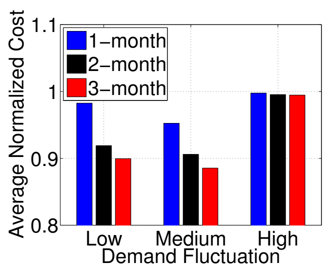


(d) Cost CDF (stable demands)

Fig. 4. Cost performance of online algorithms without *a priori* knowledge of future demands. All costs are normalized to All-on-demand.

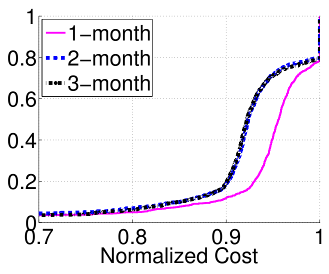


(a) Cost CDF

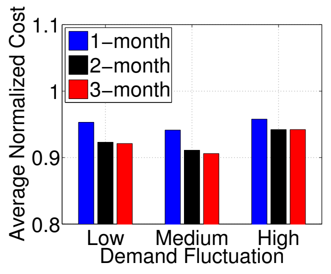


(b) Average cost in different user groups

Fig. 5. Cost performance of the deterministic algorithm with various prediction windows. All costs are normalized to the online deterministic algorithm (Algorithm 1) without any future information.



(a) Cost CDF



(b) Average cost in different user groups

Fig. 6. Cost performance of the randomized algorithm with various prediction windows. All costs are normalized to the online randomized algorithm (Algorithm 1) without any future information.

future knowledge (i.e., Algorithm 1 and 2). Figs. 5 and 6 illustrate the simulation results, where all costs are *normalized to Algorithm 1 and 2, respectively*.

As expected, the more information we know about the future demands (i.e., longer prediction window), the better the cost performance. Yet, the marginal benefits of having long-term predictions are diminishing. As shown in Figs. 5a and 6a, long prediction windows will not see proportional performance gains. This is especially the case for the randomized algorithm, in which knowing the 2-month future demand *a priori* is no different from knowing 3 months beforehand.

Also, we can see in Fig. 5b that for the deterministic algorithm, having future information only benefits those users whose demands are stable or with medium fluctuation. This is because the deterministic online algorithm is almost optimal for users with highly fluctuating demands (see Fig. 4b), leaving no space for further improvements. On the other hand, we see in Fig. 6b that the benefits of knowing future demands are consistent for all users with the randomized algorithm.

VIII. Related Work

On-demand and reserved instances are the two most prominent pricing options that are widely supported in leading IaaS clouds [3], [4], [5]. Many case studies [11] show that effectively combining the use of the two instances leads to a significant cost reduction.

There exist some works in the literature, including both algorithm design [6], [7], [24] and prototype implementation [8], focusing on combining the two instance options in a cost efficient manner. All these works assume, either explicitly or implicitly, that workloads are statistically stationary in the long-term future and can be accurately predicted *a priori*. However, it has been observed that in real production applications, ranging from enterprise applications to large e-commerce sites, workload is highly variable and statistically non-stationary [9], [10]. Furthermore, most workload prediction schemes, e.g., [25], [26], [27], are only suitable for predictions over a very short term (from half an hour to several hours). Such limitation is also shared by general predicting techniques, such as ARMA [28] and GARCH models [29]. Some long-term workload prediction schemes [30], [31], on the other hand, are reliable only when demand patterns are easy to recognize with some clear trends. Even in this case, the prediction window is at most days or weeks into the future [30], which is far shorter than the typical span of a reservation period (at least one year in Amazon EC2 [3]). All these factors significantly limit the practical use of existing works.

Our online strategies are tied to the online algorithm literature [18]. Specifically, our instance reservation problem captures a class of rent-or-buy problems, including the ski rental problem [22], the Bahncard problem [19], and the TCP acknowledgment problem [20], as special cases when a user demands no more than one instance at a time. In these problems, a customer obtains *a single item* either by paying a repeating cost (renting) per usage or by paying a one-time cost (buying) to eliminate the repeating cost. A customer makes *one-dimensional* decisions only on the timing of buying. Our problem is more complicated as a user demands *multiple instances* at a time and makes *two-dimensional* decisions on both the timing and quantity of its reservation. A similar "multi-item rent-or-buy" problem has also been investigated in [23], where a dynamic server provisioning problem is considered and an online algorithm is designed to dynamically turn on/off servers to serve time-varying workloads with a minimum energy cost. It is shown in [23] that, by dispatching jobs to servers that are idle or off the most recently, the problem reduces to a set of independent ski rental problems. Our problem does not have such a separability structure and cannot be equivalently decomposed into independent single-instance reservation (Bahncard) problems, mainly due to the possibility of time multiplexing multiple jobs on the same reserved instance. It is for this reason that the problem is challenging to solve even in the offline setting.

Besides instance reservation, online algorithms have also been applied to reduce the cost of running a file system in the cloud. The recent work [32] introduces a *constrained ski-rental problem* with extra information of query arrivals (the first or second moment of the distribution), proposing new online algorithms to achieve improved competitive ratios. [32] is orthogonal to our work as it takes advantage of additional demand information to make rent-or-buy decisions for a single item.

IX. Concluding Remarks and Future Work

Acquiring instances at the cost-optimal commitment level for time-varying workloads is critical for cost management to lower IaaS service costs. In particular, when should a user reserve instances, and how many instances should it reserve? Unlike existing reservation strategies that require knowledge of the long-term future demands, we propose two online algorithms, one deterministic and another randomized, that dynamically reserve instances without knowledge of the future demands. We show that our online algorithms incur near-optimal costs with the best possible competitive ratios, i.e., $2-\alpha$ for the deterministic algorithm and $e/(e-1+\alpha)$ for the randomized algorithm. Both online algorithms can also be easily extended to cases when short-term predictions are reliable. Large-scale simulations driven by 40 GB Google cluster-usage traces further indicate that significant cost savings are derived from our online algorithms and their extensions, under the prevalent Amazon EC2 pricing.

One of the issues that we have not discussed in this paper is the combination of different types of reserved instances with different reservation periods and utilization levels. For example, Amazon EC2 offers 1-year and 3-year reserved instances with light, medium, and high utilizations. Effectively combining these reserved instances with on-demand instances could further reduce instance acquisition costs. We note that when a user demands no more than one instance at a time and the reservation period is infinite, the problem reduces to *Multislope Ski Rental* [33]. However, it remains unclear if and how the results obtained for Multislope Ski Rental could be extended to instance acquisition with multiple reservation options.

References

[1] "Gartner Says Worldwide Cloud Services Market to Surpass $109 Billion in 2012," https://www.gartner.com/it/page.jsp?id=2163616.
[2] "The Future of Cloud Adoption," http://cloudtimes.org/2012/07/14/the-future-of-cloud-adoption/.
[3] Amazon EC2 Pricing, http://aws.amazon.com/ec2/pricing/.
[4] ElasticHosts, http://www.elastichosts.com/.
[5] GoGrid Cloud Hosting, http://www.gogrid.com.
[6] Y. Hong, M. Thottethodi, and J. Xue, "Dynamic server provisioning to minimize cost in an IaaS cloud," in *Proc. ACM SIGMETRICS*, 2011.
[7] C. Bodenstein, M. Hedwig, and D. Neumann, "Strategic decision support for smart-leasing Infrastructure-as-a-Service," in *Proc. 32nd Intl. Conf. on Info. Sys. (ICIS)*, 2011.
[8] K. Vermeersch, "A broker for cost-efficient qos aware resource allocation in EC2," Master's thesis, University of Antwerp, 2011.
[9] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 31–44, 2007.
[10] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proc. IEEE/ACM Intl. Conf. on Autonomic Computing (ICAC)*, 2010.
[11] AWS Case studies, http://aws.amazon.com/solutions/case-studies/.
[12] "Cloudability," http://cloudability.com.
[13] "Cloudyn," http://www.cloudyn.com.
[14] "Cloud Express by Apptio," https://www.cloudexpress.com.
[15] W. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality*. John Wiley and Sons, 2011.
[16] Google Cluster-Usage Traces, http://code.google.com/p/googleclusterdata/.
[17] "Amazon ElastiCache," http://cloudability.com.
[18] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
[19] R. Fleischer, "On the Bahncard problem," *Theoretical Computer Science*, vol. 268, no. 1, pp. 161–174, 2001.
[20] A. Karlin, C. Kenyon, and D. Randall, "Dynamic TCP acknowledgment and other stories about e/(e-1)," *Algorithmica*, vol. 36, no. 3, pp. 209–224, 2003.
[21] W. Wang, B. Li, and B. Liang, "To reserve or not to reserve: Optimal online multi-instance acquisition in IaaS clouds," University of Toronto, Tech. Rep., 2013. [Online]. Available: http://iqua.ece.toronto.edu/~bli/papers/onlinereserve.pdf
[22] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki, "Competitive randomized algorithms for nonuniform problems," *Algorithmica*, vol. 11, no. 6, pp. 542–571, 1994.
[23] T. Lu, M. Chen, and L. L. Andrew, "Simple and effective dynamic provisioning for power-proportional data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1611–1171, 2013.
[24] W. Wang, D. Niu, B. Li, and B. Liang, "Revenue maximization with dynamic auctions in iaas cloud markets," in *Proc. IEEE ICDCS*, 2013.
[25] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proc. USENIX NSDI*, 2008.
[26] B. Guenter, N. Jain, and C. Williams, "Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning," in *Proc. IEEE INFOCOM*, 2011.
[27] Í. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Greenhadoop: leveraging green energy in data-processing frameworks," in *Proc. ACM EuroSys*, 2012.
[28] G. Box, G. Jenkins, and G. Reinsel, *Time Series Analysis: Forecasting and Control*. Prentice Hall, 1994.
[29] T. Bollerslev, "Generalized autoregressive conditional heteroskedasticity," *Journal of Econometrics*, vol. 31, no. 3, pp. 307–327, 1986.
[30] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. IEEE/ACM Intl. Conf. on Autonomic Computing (ICAC)*, 2005.
[31] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Capacity management and demand prediction for next generation data centers," in *Proc. IEEE Intl. Conf. on Web Services (ICWS)*, 2007.
[32] A. Khanafer, M. Kodialam, and K. P. N. Puttaswamy, "The constrained ski-rental problem and its application to online cloud cost optimization," in *Proc. IEEE INFOCOM*, 2013.
[33] Z. Lotker, B. Patt-Shamir, D. Rawitz, S. Albers, and P. Weil, "Rent, lease or buy: Randomized algorithms for multislope ski rental," in *Proc. Symp. on Theoretical Aspects of Computer Science (STAC)*, 2008.

Elasticity in Cloud Computing: What It Is, and What It Is Not

Nikolas Roman Herbst, Samuel Kounev, Ralf Reussner
Institute for Program Structures and Data Organisation
Karlsruhe Institute of Technology
Karlsruhe, Germany
{herbst, kounev, reussner}@kit.edu

Abstract

Originating from the field of physics and economics, the term elasticity is nowadays heavily used in the context of cloud computing. In this context, elasticity is commonly understood as the ability of a system to automatically provision and deprovision computing resources on demand as workloads change. However, elasticity still lacks a precise definition as well as representative metrics coupled with a benchmarking methodology to enable comparability of systems. Existing definitions of elasticity are largely inconsistent and unspecific, which leads to confusion in the use of the term and its differentiation from related terms such as scalability and efficiency; the proposed measurement methodologies do not provide means to quantify elasticity without mixing it with efficiency or scalability aspects. In this short paper, we propose a precise definition of elasticity and analyze its core properties and requirements explicitly distinguishing from related terms such as scalability and efficiency. Furthermore, we present a set of appropriate elasticity metrics and sketch a new elasticity tailored benchmarking methodology addressing the special requirements on workload design and calibration.

1 Introduction

Elasticity has originally been defined in physics as a material property capturing the capability of returning to its original state after a deformation. In economical theory, informally, elasticity denotes the sensitivity of a dependent variable to changes in one or more other variables [1]. In both cases, elasticity is an intuitive concept and can be precisely described using mathematical formulas.

The concept of elasticity has been transferred to the context of cloud computing and is commonly considered as one of the central attributes of the cloud paradigm [10]. For marketing purposes, the term elasticity is heavily used in cloud providers' advertisements and even in the naming of specific products or services. Even though tremendous efforts are invested to enable cloud systems to behave in an elastic manner, no common and precise understanding of this term in the context of cloud computing has been established so far, and no ways have been proposed to quantify and compare elastic behavior. To underline this observation, we cite five definitions of elasticity demonstrating the inconsistent use and understanding of the term:

1. *ODCA, Compute Infrastructure-as-a-Service* [9] "[...] defines elasticity as the configurability and expandability of the solution [...] Centrally, it is the ability to scale up and scale down capacity based on subscriber workload."
2. *NIST Definition of Cloud Computing* [8] "Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time."
3. *IBM, Thoughts on Cloud, Edwin Schouten, 2012* [11] "Elasticity is basically a 'rename' of scalability [...]" and "removes any manual labor needed to increase or reduce capacity."
4. *Rich Wolski, CTO, Eucalyptus, 2011* [12] "Elasticity measures the ability of the cloud to map a single user request to different resources."
5. *Reuven Cohen, 2009* [2] Elasticity is "the quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources."

Definitions (1), (2), and (3) in common describe elasticity as the scaling of system resources to increase or decrease capacity, whereby definitions (1), (2) and (5) specifically state that the amount of provisioned resources is somehow connected to the recent demand or workload. In these two points there appears to be some consent. Definitions (4) and (5) try to capture elasticity in a generic way as a 'quantifiable' system ability to handle requests using different resources. Both of these definitions, however, neither give concrete details on the core aspects of elasticity, nor provide any hints on how elasticity can be measured. Definition (3) assumes that no manual work at all is needed, whereas in the NIST definition (2), the processes enabling elasticity do not need to be fully automatic. In addition, the NIST definition adds the adjective 'rapid' to elasticity and draws the idealistic picture of 'perfect' elasticity where endless resources are available with an appropriate provisioning at any point in time, in a way that the end-user does not experience any performance variability.

We argue that existing definitions of elasticity fail to capture the core aspects of this term in a clear and unambiguous manner and are even contradictory in some parts. To address this issue, in this short paper, we propose a new refined definition of elasticity considering in detail its core aspects and the prerequisites of elastic system behavior (Section 2). Thereby, we clearly differentiate elasticity from its related terms scalability and efficiency. In Section 4, we present metrics that are able to capture elasticity, followed by Section 5, in which we outline a benchmarking methodology for quantifying elasticity discussing the issues of representativeness, reproducibility and fairness of the measurement approach.

2 Elasticity

In this section, we first describe some important prerequisites in order to be able to speak of elasticity, present a new refined and comprehensive definition, and then analyse its core aspects and dimensions. Finally, we differentiate between elasticity and its related terms scalability and efficiency.

2.1 Prerequisites

The scalability of a system including all hardware, virtualization, and software layers within its boundaries is a prerequisite in order to be able to speak of elasticity. Scalability is the ability of a system to sustain increasing workloads with adequate performance provided that hardware resources are added. Scalability in the context of distributed systems has been defined in [6], as well as more recently in [3, 4], where also a measurement methodology is proposed.

Given that elasticity is related to the ability of a system to adapt to changes in workloads and resource demands, the existence of at least one specific adaptation process is assumed. The latter is normally automated, however, in a broader sense, it could also contain manual steps. Without a defined adaptation process, a scalable system cannot behave in an elastic manner, as scalability on its own does not include temporal aspects.

When evaluating elasticity, the following points need to be checked beforehand:

- *Autonomic Scaling:*
  What adaptation process is used for autonomic scaling?
- *Elasticity Dimensions:*
  What is the set of resource types scaled as part of the adaptation process?
- *Resource Scaling Units:*
  For each resource type, in what unit is the amount of allocated resources varied?
- *Scalability Bounds:*
  For each resource type, what is the upper bound on the amount of resources that can be allocated?

2.2 Definition

Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources *match* the current demand as closely as possible.

2.3 Dimensions and Core Aspects

Any given adaptation process is defined in the context of at least one or possibly multiple types of resources that can be scaled up or down as part of the adaptation. Each resource type can be seen as a separate dimension of the adaptation process with its own elasticity properties. If a resource type is a container of other resources types, like in the case of a virtual machine having assigned CPU cores and RAM, elasticity can be considered at multiple levels. Normally, resources of a given resource type can only be provisioned in discrete units like CPU cores, virtual machines (VMs), or physical nodes. For each dimension of the adaptation process with respect to a specific resource type, elasticity captures the following core aspects of the adaptation:

Speed The speed of scaling up is defined as the time it takes to switch from an underprovisioned state to an optimal or overprovisioned state. The speed of scaling down is defined as the time it takes to switch from an overprovisioned state to an optimal or underprovisioned state. The speed of scaling up/down does not correspond directly to the technical resource provisioning/deprovisioning time.

Precision The precision of scaling is defined as the absolute deviation of the current amount of allocated resources from the actual resource demand.

As discussed above, elasticity is always considered with respect to one or more resource types. Thus, a direct comparison between two systems in terms of elasticity is only possible if the same resource types (measured in identical units) are scaled.

To evaluate the actual observable elasticity in a given scenario, as a first step, one must define the criterion based on which the amount of provisioned resources is considered to *match* the actual current demand needed to satisfy the system's given performance requirements. Based on such a matching criterion, specific metrics that quantify the above mentioned core aspects, as discussed in more detail in Section 4, can be defined to quantify the practically achieved elasticity in comparison to the hypothetical *optimal elasticity*. The latter corresponds to the hypothetical case where the system is scalable with respect to all considered elasticity dimensions without any upper bounds on the amount of resources that can be provisioned and where resources are provisioned and deprovisioned immediately as they are needed exactly matching the actual demand at any point in time. *Optimal elasticity*, as defined here, would only be limited by the resource scaling units.

2.4 Differentiation

In this section, we highlight the conceptual differences between elasticity and the related terms scalability and efficiency.

Scalability is a prerequisite for elasticity, but it does not consider temporal aspects of how fast, how often, and at what granularity scaling actions can be performed. Scalability is the ability of the system to sustain increasing workloads by making use of additional resources, and therefore, in contrast to elasticity, it is not directly related to how well the actual resource demands are matched by the provisioned resources at any point in time.

Efficiency expresses the amount of resources consumed for processing a given amount of work. In contrast to elasticity, efficiency is not limited to resource types that are scaled as part of the system's adaptation mechanisms. Normally, better elasticity results in higher efficiency. The other way round, this implication is not given, as efficiency can be influenced by other factors independent of the system's elasticity mechanisms (e.g., different implementations of the same operation).

3 Derivation of the Matching Function

To capture the criterion based on which the amount of provisioned resources is considered to match the actual current demand, we define a matching function $m(w) = r$ as a system specific function that returns the minimal amount of resources r for a given resource type needed to satisfy the system's performance requirements at a specified workload intensity. The workload intensity w can be specified either as the number of workload units (e.g., user requests) present at the system at the same time (concurrency level), or as the number of workload units that arrive per unit of time (arrival rate). A matching function is needed for both directions of scaling (up/down), as it cannot be assumed that the optimal resource allocation level when transitioning from an underprovisioned state (upwards) are the same as when transitioning from an overprovisioned state (downwards).

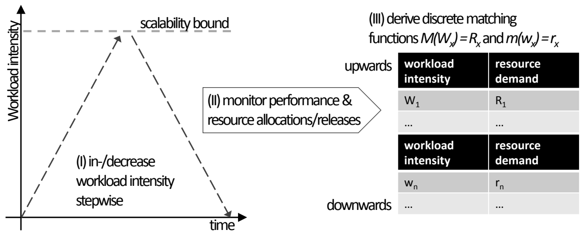

Workload intensity
scalability bound
(I) in-/decrease workload intensity stepwise
time
(II) monitor performance & resource allocations/releases
(III) derive discrete matching functions $M(W_x) = R_x$ and $m(w_x) = r_x$
upwards
downwards
workload intensity
resource demand
W_1
R_1
...
workload intensity
resource demand
w_n
r_n
...

Figure 1: Illustration of a Measurement-based Derivation of Matching Functions

The matching functions can be derived based on measurements, as illustrated in Figure 1, by increasing the workload intensity w stepwise, while measuring the resource consumption r, and tracking resource allocation changes. The process is then repeated for decreasing w. After each change in the workload intensity, the system should be given enough time to adapt its resource allocations reaching a stable state for the respective workload intensity. As a rule of thumb, at least two times the technical resource provisioning time is recommended to use as a minimum. As a result of this step, a system specific table is derived that maps workload intensity levels to resource demands, and the other way round, for both scaling directions within the scaling bounds.

4 Elasticity Metrics

To capture the core elasticity aspects *speed* and *precision*, we propose the following definitions and metrics as illustrated in Figure 2:

- $\overline{A}$ is the average time to switch from an underprovisioned state to an optimal or overprovisioned state and corresponds to the average *speed* of scaling up.

- $\sum A$ is the accumulated time in underprovisioned state.
- $\overline{U}$ is the average amount of underprovisioned resources during an underprovisioned period.
- $\sum U$ is the accumulated amount of underprovisioned resources.
- $\overline{B}$, $\sum B$, $\overline{O}$, and $\sum O$ are defined similarly for overprovisioned states.

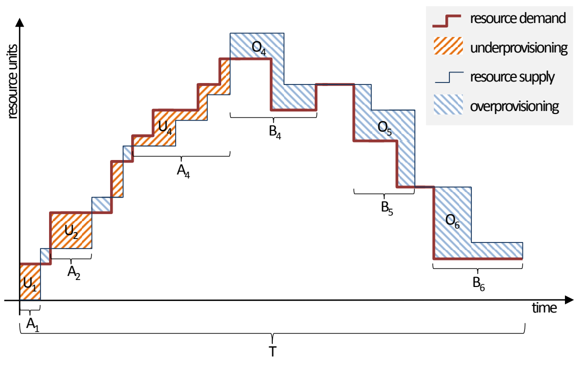


Figure 2: Capturing Core Elasticity Metrics

We define the average *precision* of scaling up P_u as $P_u = \frac{\sum U}{T}$ where T is the total duration of the evaluation period, and accordingly $P_d = \frac{\sum O}{T}$ is defined as the average precision of scaling down. Based on the above defined quantities, one could define an *elasticity* metric for scaling up E_u as inversely proportional to $\overline{A}$ and $\overline{U}$, e.g. $E_u = \frac{1}{\overline{A} \times \overline{U}}$, and accordingly *elasticity* for scaling down $E_d = \frac{1}{\overline{B} \times \overline{O}}$. The elasticity of a system under test (SUT) s can then be captured in a matrix M_s where each vector v_d represents an elasticity dimension d and contains the values of the elasticity core metrics E_u, $\overline{A}$, P_u for scaling up and E_d, $\overline{B}$, P_d for scaling down.

As an alternative to these metrics, the dynamic time warping (DTW) distance [7] can be used as a robust distance metric to capture the similarity between the demand and supply curves as well as to approximate the technical reaction time of the adaptation mechanism. A case study demonstrating this approach can be found in [5].

5 Towards Benchmarking Elasticity

Characterizing the elasticity of a single system is not a simple task on its own and it becomes even more complicated when comparing different systems. An elasticity benchmark is expected to deliver reproducible results and generate a consistent order of the different systems under test (SUTs) reflecting their potential and observed elasticity, while not mixing this with general system efficiency and scalability aspects. Traditional benchmarking approaches induce identical workloads on different SUTs to provide a basis for fair comparisons, whereas an elasticity benchmark is required to induce identical demand curves. If two elastic systems exhibit significant differences in efficiency (the amount of resources required for meeting performance requirements at a given workload intensity level), it might well be that when processing an identical workload, their adaptation mechanisms are exercised in a significantly different manner. As illustrated in Figure 3, in that case, deriving the elasticity metrics for the same workload would result in unfair comparison since the more efficient system would appear to exhibit better elasticity given that its adaptation mechanisms were not stressed to the same extent.

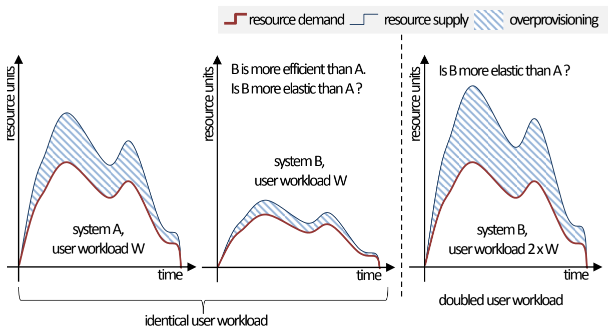

resource demand
resource supply
overprovisioning
resource units
system A, user workload W
time
B is more efficient than A.
Is B more elastic than A ?
system B, user workload W
Is B more elastic than A ?
system B, user workload 2 x W
identical user workload
doubled user workload

Figure 3: Elasticity vs. Efficiency

Therefore, the first step towards portability of an elasticity benchmark and comparability of its results would be the specification of a representative set of demand curves and common performance goals in terms of responsiveness, throughput or utilisation for the considered resource types. The demand curves themselves should contain bursts of different intensity, upward and downward scaling trends and seasonal patterns of different shapes, concerning amplitude, duration and base level capturing the most representative real-life scenarios. Further challenges include the automated derivation of the mapping functions as well as the generation of a workload that induces the targeted demand curves as accurately as possible on the evaluated SUTs.

6 Conclusion

In this short paper, we proposed a refined definition of elasticity to contribute in establishing a common understanding of this term in the context of cloud computing. Furthermore, we examined the core aspects of elasticity explicitly differentiating it conceptually from the classical notions of scalability and efficiency. Finally, we propose metrics to capture the core elasticity aspects as well as an elasticity benchmarking approach focusing on the special requirements on workload design and its implementation.

References

[1] CHIANG, A. C., AND WAINWRIGHT, K. *Fundamental methods of mathematical economics*, 4. ed., internat. ed., [repr.] ed. McGraw-Hill [u.a.], Boston, Mass. [u.a.], 2009.

[2] COHEN, R. Defining Elastic Computing, September 2009. `http://www.elasticvapor.com/2009/09/defining-elastic-computing.html`, last consulted Feb. 2013.

[3] DUBOC, L. *A Framework for the Characterization and Analysis of Software Systems Scalability*. PhD thesis, Department of Computer Science, University College London, 2009. `http://discovery.ucl.ac.uk/19413/1/19413.pdf`.

[4] DUBOC, L., ROSENBLUM, D., AND WICKS, T. A Framework for Characterization and Analysis of Software System Scalability. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)* (2007), ACM, pp. 375–384.

[5] HERBST, N. R. Quantifying the Impact of Configuration Space for Elasticity Benchmarking. Study thesis, Faculty of Computer Science, Karlsruhe Institute of Technology (KIT), Germany, 2011. `http://sdqweb.ipd.kit.edu/publications/pdfs/Herbst2011a.pdf`.

[6] JOGALEKAR, P., AND WOODSIDE, M. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems 11* (2000), 589–603.

[7] KEOGH, E., AND RATANAMAHATANA, C. A. Exact indexing of dynamic time warping. *Knowl. Inf. Syst. 7*, 3 (Mar. 2005), 358–386.

[8] MELL, P., AND GRANCE, T. The NIST Definition of Cloud Computing. Tech. rep., U.S. National Institute of Standards and Technology (NIST), 2011. Special Publication 800-145, `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[9] OCDA. Master Usage Model: Compute Infratructure as a Service. Tech. rep., Open Data Center Alliance (OCDA), 2012. `http://www.opendatacenteralliance.org/docs/ODCA_Compute_IaaS_MasterUM_v1.0_Nov2012.pdf`.

[10] PLUMMER, D. C., SMITH, D. M., BITTMAN, T. J., CEARLEY, D. W., CAPPUCCIO, D. J., SCOTT, D., KUMAR, R., AND ROBERTSON, B. Study: Five Refining Attributes of Public and Private Cloud Computing. Tech. rep., Gartner, 2009. `http://www.gartner.com/DisplayDocument?doc_cd=167182`, last consulted Feb. 2013.

[11] SCHOUTEN, E. Rapid Elasticity and the Cloud, September 2012. `http://thoughtsoncloud.com/index.php/2012/09/rapid-elasticity-and-the-cloud/`, last consulted Feb. 2013.

[12] WOLSKI, R. Cloud Computing and Open Source: Watching Hype meet Reality, May 2011. `http://www.ics.uci.edu/~ccgrid11/files/ccgrid-11_Rich_Wolsky.pdf`, last consulted Feb. 2013.

K-Scope: Online Performance Tracking for Dynamic Cloud Applications

Li Zhang Xiaoqiao Meng Shicong Meng Jian Tan
IBM TJ Watson Research Center
{*zhangli, xmeng, smeng, tanji*}*@us.ibm.com*

1 Introduction

Cloud computing is an ongoing technology evolution that reshapes every aspects of computing. Cloud provides on-demand, flexible and easy-to-use resource provisioning. It is also an open platform where Cloud users can share software components, resources and services. These features give rise to several emerging Cloud application development and deployment paradigms, represented by continuous delivery and shared platform services.

Continuous Delivery [3], coined by Amazon, is a new way of releasing software wherein a Cloud application (e.g., Amazon web services) is delivered through frequent incremental updates. Cloud enables this paradigm by allowing developers to easily create a pipeline of automated application building, testing and deployment. For instance, application developers can quickly produce multiple application deployment for different development stages through virtual machine replication. Continuous delivery provides tremendous benefits in improving user experience and reduces the risk of each individual release substantially.

Shared Platform Services are commonly used in Cloud applications, and rapidly gaining popularity with increasing Platform-as-a-Service offers from Cloud service providers. Perhaps the most widely used platform service today is database or datastore services (e.g., SimpleDB from Amazon and Cloud SQL from Google) which are large-scale multi-tenant databases or datastores shared by multiple Cloud applications through a set of data access APIs. Enterprise users sometimes also deploy their own database/datastore servers shared by multiple applications in their virtual private Cloud (VPC). These shared data services reduce the management burden for application developers.

Despite the enormous convenience and great potential of these new paradigms, they also introduce new performance management challenges due to the volatility embedded in these techniques as well as the lack of well-defined performance requirements. For instance, updates in continuous deployment often change the behavior and the performance characteristics of an application, which may lead to performance degradation and service level agreement (SLA) violations. Similarly, due to the sharing nature of data services, one may experience fluctuation in data access performance when the overall workloads of the data service change. We refer to Cloud applications utilizing these features as *dynamic Cloud applications* to distinguish them from applications using traditional development life cycle and dedicated software components.

These challenges call for a fundamental piece missing from today's Cloud services, that is the ability to *continuously*, *efficiently* and *accurately* capture the most up-to-date performance characteristics of a dynamic Cloud application. Existing performance modeling approaches, however, do not readily provide this continuous modeling ability, primarily because they are designed with a traditional static deployment in mind where an application runs on dedicated machines and its implementation does not change during the modeling process. Some of them [9] must run offline with long model training time and high cost. Others [8, 12, 10] cannot explicitly model multiple request types or multiple functional layers which are common for Cloud applications. There are also techniques [1, 2] that can capture performance changes at different functional layers, but require instrumentation of the application.

In this paper, we introduce the first online, multi-request, multi-layer application performance modeling approach. It is non-intrusive in the sense that it infers critical performance model metrics such as request service time at different functional layers (e.g, web/application/database servers), which are usually unobservable, only from basic monitoring information such as end-to-end response time and CPU utilization, without instrumenting applications. Furthermore, it utilizes

Kalman filters [7] to continuously adjust model metrics to keep the model consistent with the dynamic Cloud application. As a result, an up-to-date performance model is always ready for users to query and perform tasks such as capacity planning and auto-scaling. For instance, it can quantitatively predict how much resources are needed at different functional layers to maintain a given performance, even when the application is constantly undergoing software updates.

2 Approach Overview

We consider a Cloud application consisting of multiple functional layers, e.g., web server layer, application server layer and database server layer. Such an application processes a number of different types of requests, each of which can be quite different in terms of execution time and resource consumption. Furthermore, the application has a targeted performance goal or service level agreement (SLA), e.g., the average response time should be smaller than 500ms. We assume the available monitoring data for these Cloud applications are basic system utilization metrics (e.g., CPU utilization), throughput and response time. These information are readily available on most Cloud platforms [6].

We choose to use non-intrusive modeling techniques that provide an easy-to-use performance model that can predict application resource utilization and performance, rather than using instrumentation based tracing techniques. Specifically, we use the queuing network model as the basic framework as it is general enough to model multi-layer multi-request applications. To cope with the changing performance characteristics in dynamic Cloud applications, in particular, the request service time which is the time a server spent to process a request, we need an agile, online model parameter estimation technique, rather than traditional constrained optimization based offline estimation techniques. Kalman filter, as a time-tested technique for estimating potentially changing future states, falls nicely into our design.

2.1 Queueing Network Model

Queueing network models are commonly used to capture the performance of complex computer systems [4]. They have been shown to provide accurate characterization of request level and system level performance metrics [5, 11]. Well calibrated queueing network models are the basis for performance sizing and capacity planning. Here we also use a general queueing network model for Cloud applications.

We will use a 3-class, 2-tier system to illustrate our performance modeling and tracking methodology. It can easily be extended to a general n class k tier system. We first define a set of variables for the model:

$$\begin{aligned}
\lambda_i &= \text{Arrival rate of class } i \text{ jobs.}\\
S_{ij} &= \text{Average service time of class } i \text{ jobs at tier } j.\\
d_i &= \text{Additional delay for class } i \text{ jobs in system.}\\
u_{0j} &= \text{Background utilization for tier } j.\\
u_j &= \text{Average utilization for tier } j.\\
R_i &= \text{Average response time for class } i \text{ jobs in system.}
\end{aligned}$$

Under appropriate assumptions, the system performance and resource utilization can be approximated by the queueing analytic relations below.

$$u_j = u_{0j} + \lambda_1 S_{1j} + \lambda_2 S_{2j} + \lambda_3 S_{3j}, j \in \{1,2\} \quad (1)$$

$$R_i = d_i + \frac{S_{i1}}{1-u_1} + \frac{S_{i2}}{1-u_2}, i \in \{1,2,3\} \quad (2)$$

In vector form: $\mathbf{z} := (u_1, u_2, R_1, R_2, R_3)^T = \mathbf{h}(\mathbf{x})$. The assumptions for the above formulate to hold are quite general. For example, under Poisson arrivals and processor sharing policy at each server, the formulate are exact. Processor sharing policy can reasonably approximate the scheduling behaviors in modern operating systems. Numerous studies have demonstrated that the queueing model above provides a good approximation to the real system.

It is relatively easy to measure the aggregate system utilization u_1, u_2, the request throughput $\lambda_1, \lambda_2, \lambda_3$, and the end-to-end response times R_1, R_2, R_3. The delay and service time parameters, however, are very difficult to measure directly. These parameters are the key quantitative information of the system model. In our 3-class 2-server example, the system parameters are

$$\mathbf{x} = (u_{01}, u_{02}, d_1, d_2, d_3, S_{11}, S_{21}, S_{31}, S_{12}, S_{22}, S_{32})^T \quad (3)$$

an 11-dimension vector. The important problem we need to solve now is to estimate the system parameters $\mathbf{x}$ based on the measurement data $\mathbf{z} = (u_1, u_2, R_1, R_2, R_3)^T$. The off-line parameter estimation problem has been addressed in [11] by formulating the problem as an optimization problem.

Below we address this on-line parameter estimation problem with noisy measurement data. The challenge is how to efficiently and accurately estimate $\mathbf{x}$ on line from a continuous stream of measurements $\mathbf{z}$. Kalman filter theory is a perfect tool to tackle this problem.

2.2 Kalman Filter

Kalman filter is developed by Rudolf E. Kalman around 1960. It is commonly used to estimate the values of hidden state variables of a dynamic system that is excited by stochastic disturbances and stochastic measurement noise. In real systems, all the variables are functions of time. Measurements will change over time. Parameter values will have estimates that are updated over time. The dynamics of the system following the Kalman filter framework is

$$\mathbf{x}(t) = \mathbf{F}(t)\mathbf{x}(t-1)+\mathbf{w}(t) = \mathbf{x}(t-1)+\mathbf{w}(t), \quad (4)$$
$$\mathbf{z}(t) = \mathbf{H}(t)\mathbf{x}(t-1)+\mathbf{v}(t). \quad (5)$$

Here $\mathbf{x}$ is the state variable that is not observed. $F(t)$ is the state transition model that describes the evolution of the state over time. $\mathbf{w}(t)$ is the process noise which is assumed to be a zero mean, multi-variate Normal distribution with certain covariance matrix $\mathscr{Q}(t)$, i.e. $\mathbf{w}(t) \sim \mathscr{N}(0,\mathscr{Q}(t))$. $\mathbf{z}(t)$ is the measurement vector. $\mathbf{H}(t)$ is the observation model which maps the true state space into the observed space. $v(t)$ is the observation noise which is assumed to be a zero mean, multi-variate Normal distribution with certain covariance matrix $\mathscr{R}(t)$, i.e. $\mathbf{v}(t) \sim \mathscr{N}(0,\mathscr{R}(t))$. The covariance matrices $\mathscr{Q}$ and $\mathscr{R}$ are not directly measurable. They will be tuned based on best practice heuristics.

Since the measurement model is a non-linear function of the system state parameters (due to the utilization u in the denominator), we must use the 'Extended' version of the Kalman filter. $\mathbf{H}(t)$ is computed as, $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\mathbf{x}(t))$ Since we don't really know $\mathbf{x}$ at time t, we will estimate it based on all the information we have before time t. $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\hat{\mathbf{x}}(t|t-1))$ Here $\hat{\mathbf{x}}(t|t-1)$ is the estimate of $\mathbf{x}(t)$ given all the information up to time $t-1$.

The state of the filter is represented by two variables:

- $\hat{\mathbf{x}}(t|t)$ is the estimate of state at time t given observations up to and including time t.
- $\mathbf{P}(t|t)$ is the error covariance matrix (a quantitative measure of estimated accuracy of the state estimate).

Here are the two sets of equations for the Kalman filter algorithm:

Predict:

$$\hat{\mathbf{x}}(t|t-1) = \mathbf{F}(t)\hat{\mathbf{x}}(t-1|t-1) \quad (6)$$
$$\mathbf{P}(t|t-1) = \mathbf{F}(t)\mathbf{P}(t-1|t-1)\mathbf{F}^T(t)+\mathscr{Q}(t) \quad (7)$$

Update:

$$\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\hat{\mathbf{x}}(t|t-1)) \quad (8)$$
$$\mathbf{S}(t) = \mathbf{H}(t)\mathbf{P}(t|t-1)\mathbf{H}^T(t)+\mathscr{R}(t) \quad (9)$$
$$\mathbf{K}(t) = \mathbf{P}(t|t-1)\mathbf{H}^T(t)\mathbf{S}^{-1}(t) \quad (10)$$
$$\hat{\mathbf{x}}(t|t) = \hat{\mathbf{x}}(t|t-1)+\mathbf{K}(t)(\mathbf{z}(t)-\mathbf{h}(\hat{\mathbf{x}}(t|t-1))) \quad (11)$$
$$\mathbf{P}(t|t) = (\mathbf{I}-\mathbf{K}(t)\mathbf{H}(t))\mathbf{P}(t|t-1) \quad (12)$$

In our 3-class 2-server queueing network example, the Jacobian is given by, $\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{bmatrix} J_{11} & J_{12} & J_{13} & J_{14} \\ J_{21} & J_{22} & J_{23} & J_{24} \end{bmatrix}$ The algorithm iterates between the predict and update steps as new measurement data arrives.

2.3 Applications

K-Scope has a wide range of applications, including performance diagnosis, answering what-if queries, capacity planning and performance-driven dynamic provisioning.

Performance Diagnosis. Performance diagnosis for multi-layer applications is painful as generic monitoring provides only end-to-end performance statistics which offer little insight on the performance of individual functional layers. K-Scope explicitly estimates request service time at different layers, and provides a clear breakdown of the response time.

Answering What-If Queries. A simple approach is that we first apply the model to track the system in a stable period; with all the model parameters estimated, the question can be generally solved by varying certain parameters and re-calculate the other parameters.

Capacity Planning. K-Scope also simplifies capacity planning as application developers can leverage the performance model produced by K-Scope to virtually explore a large number of deployment options and predict the corresponding performance.

Dynamic Provisioning. As K-Scope provides a breakdown of request execution time at different layers, it can guide dynamic provisioning to the bottlenecked layer. In addition, dynamic provisioning can query K-Scope to find out how many additional virtual instances are needed to maintain the targeted performance, and quickly adds the required number of instances in a single batch to minimize the window of performance violation.

3 Evaluation

We apply K-Scope to a real-world multi-layer application. In addition, we describe a simple usage scenario in which the model is used for capacity planning.

The tested workload is SOABench, an IBM internal benchmark widely used to measure the performance of Web servers. Our testbed consists of a client and a server machine. Each machine is equipped with an Intel 1.6GHz 8-core Xeon processor. The client machine runs the SOABench workload generator, a Java program that could spawn multiple threads to simulate concurrent Web service users. The server machine runs IBM WAS(WebSphere Application Server). Each Java thread in the workload generator sends a service request to the WAS server. Upon receiving the response, the thread continues to send another request. Three types of service requests are sent by the generator: for Type 1, both the request and the response have 3K Byte payload. Type 2 and 3 have 10K and 1M Byte payload respectively. This SOABench testbed follows the three-class two-tier model in the previous sections.

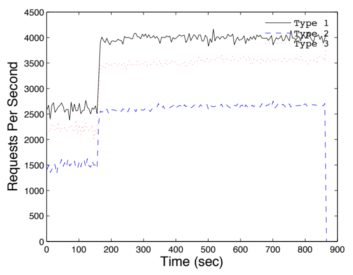
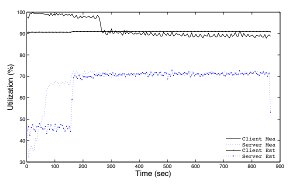
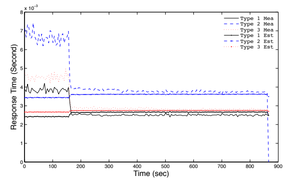


Figure 1: Workload charasterics in SOABench testbed



(a) CPU utilizations

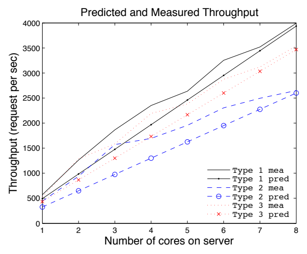


(b) Response time

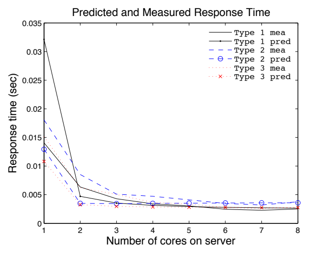
Figure 2: Measured and Predicted Results

In our first experiment, we want to use the model to track the performance of SOABench when the system resources are close to full utilization. To this end, we increase the number of threads on the client until either the client or the server has a saturated CPU usage. At this saturation point, the workload generator spawns 24 threads: 8 threads for each request type. Figure 1 shows the throughput for each request type. We run the experiment for about 15 minutes. The first 3 minutes are a warm-up period. After the warm-up, all the performance metrics become stable. We then collect data for the observable performance metrics, feed the data to the proposed model, and measure the model accuracy by comparing the predicted CPU utilization and response time to their actual values. Figure 2(a) compares the measured and estimated CPU utilization. Figure 2(b) compares the measured and the estimated response time for each request type. All these comparisons clearly show that the model can precisely track the performance.

Now we describe a case in which the model is used to address a simple capacity planning issue. The WAS server has eight cores and all these cores are dedicated to the WAS application. If the server allocates fewer cores to the WAS, how will this impact the throughput and response time? Such a typical *what-if* question can be easily answered by applying the model. In principle, if fewer CPU cores are allocated to a task, the task processing time should increase. We approximately assume that if the allocated core number on the server is reduced to $\frac{1}{x}$ of the original core number, the service time for each request, namely, S_{12}, S_{22} and S_{32}, should be multiplied by x respectively. Now if the server keeps the same utilization ratio, according to Equation (1), λ_1, λ_2 and λ_3 are reduced to $\frac{1}{x}$ of their original values respectively. After computing the adjusted u_1 from (1), we can further compute the new response time from (2). To evaluate the accuracy of this simple computation, we vary the allocated core number on the server from 1 to 8, and for each setting, we restart the WAS server. Figure 3(a) compares the computed throughput and the actual measurements. Figure 3(b) compares the response time. On both aspects, the estimation follows the ground truth.



(a) Predicted throughput when reducing CPU cores on server



(b) Predicting response time when reducing CPU cores on server

Figure 3: Application in Capacity Planning

References

[1] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Online modelling and performance-aware systems. In *HotOS* (2003), pp. 85–90.

[2] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[3] HUMBLE, J., AND FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[4] KLEINROCK, L. *Queueing Systems*. Wiley, 1976.

[5] KUMAR, D., OLSHEFSKI, D. P., AND ZHANG, L. Connection and performance model driven optimization of pageview response time. In *MASCOTS* (2009), IEEE, pp. 1–10.

[6] MENG, S., WANG, T., AND LIU, L. Monitoring continuous state violation in datacenters: Exploring the time dimension. In *ICDE* (2010), pp. 968–979.

[7] SIMON, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley, 2006.

[8] SOLOMON, B., IONESCU, D., LITOIU, M., AND MIHAESCU, M. A real-time adaptive control of autonomic computing environments. In *CASCON* (2007), pp. 124–136.

[9] URGAONKAR, B., PACIFICI, G., SHENOY, P. J., SPREITZER, M., AND TANTAWI, A. N. Analytic modeling of multitier internet applications. *TWEB 1*, 1 (2007).

[10] WOODSIDE, C. M., ZHENG, T., AND LITOIU, M. Service system resource management based on a tracked layered performance model. In *ICAC* (2006), pp. 175–184.

[11] ZHANG, L., XIA, C. H., SQUILLANTE, M. S., AND III, W. N. M. Workload service requirements analysis: A queueing network optimization approach. In *MASCOTS* (2002), IEEE.

[12] ZHENG, T., WOODSIDE, C. M., AND LITOIU, M. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng. 34*, 3 (2008), 391–406.

Adaptive Performance-Aware Distributed Memory Caching

Jinho Hwang and Timothy Wood
The George Washington University

Abstract

Distributed in-memory caching systems such as memcached have become crucial for improving the performance of web applications. However, memcached by itself does not control which node is responsible for each data object, and inefficient partitioning schemes can easily lead to load imbalances. Further, a statically sized memcached cluster can be insufficient or inefficient when demand rises and falls. In this paper we present an automated cache management system that both intelligently decides how to scale a distributed caching system and uses a new, adaptive partitioning algorithm that ensures that load is evenly distributed despite variations in object size and popularity. We have implemented an adaptive hashing system[1] as a proxy and node control framework for memcached, and evaluate it on EC2 using a set of realistic benchmarks including database dumps and traces from Wikipedia.

1 Introduction

Many enterprises use cloud infrastructures to deploy web applications that service customers on a wide range of devices around the world. Since these are generally customer-facing applications on the public internet, they feature unpredictable workloads, including daily fluctuations and the possibility of flash crowds. To meet the performance requirements of these applications, many businesses use in-memory distributed caches such as memcached to store their content. Memcached shifts the performance bottleneck away from databases by allowing small, but computationally expensive pieces of data to be cached in a simple way. This has become a key concept in many highly scalable websites; for example, Facebook is reported to use more than ten thousand memcached servers.

[1] Our system can be found in https://github.com/jinho10 as an open source project.

Large changes in workload volume can cause caches to become overloaded, impacting the performance goals of the application. While it remains common, overprovisoining the caching tier to ensure there is capacity for peak workloads is a poor solution since cache nodes are often expensive, high memory servers. Manual provisioning or simple utilization based management systems such as Amazon's AutoScale feature are sometimes employed [7], but these do not intelligently respond to demand fluctuations, particularly since black-box resource management systems often cannot infer memory utilization information.

A further challenge is that while memcached provides an easy to use distributed cache, it leaves the application designer responsible for evenly distributing load across servers. If this is done inefficiently, it can lead to cache hotspots where a single server is selected to host a large set of popular data while others are left lightly loaded. Companies such as Facebook have developed monitoring systems to help administrators observe and manage the load on their memcached servers [16, 18], but these approaches still rely on expert knowledge and manual intervention.

We have developed *adaptive hashing* that is a new adaptive cache partitioning and replica management system that allows an in-memory cache to autonomically adjust its behavior based on administrator specified goals. Compared to existing systems, our work provides the following benefits:

- A hash space allocation scheme that allows for targeted load shifting between unbalanced servers.
- Adaptive partitioning of the cache's hash space to automatically meet hit rate and server utilization goals.
- An automated replica management system that adds or removes cache replicas based on overall cache performance.

We have built a prototype system on top of the popular moxi + memcached platform, and have thoroughly evaluated its performance characteristics using real content and access logs from Wikipedia. Our results show that when system configurations are properly set, our system improves the average user reponse time by 38%, and hit rate by 31% compared to the current approaches.

2 Background and Motivation

Consistent hashing [10] has been widely used in distributed hash tables (DHT) to allow dynamically changing the number of storage nodes without having to reorganize all the data, which would be disastrous to application performance. Figure 1 illustrates basic operations of a consistent hashing scheme: node allocation, virtual nodes, and replication. Firstly, with an initial number of servers, consistent hashing calculates the hash values of each server using a hash function (such as md5 in the moxi proxy for memcached). Then, according to the predefined number of virtual nodes, the address is concatenated with "-X", X is the incremental number from 1 to number of virtual nodes. Virtual nodes are used to distribute the hash space over the number of servers. This way is particularly not efficient because the hash values of server addresses are not guaranteed to be evenly distributed over the hash space, which makes imbalances. This inefficiency is shown in Section 4.2.

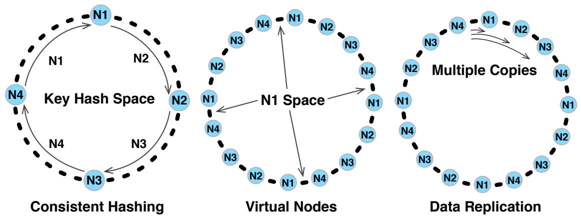


Figure 1: Consistent Hashing Operations; N_i is i^{th} cache node. Integer (32 bits) hash space consists of 2^{32} possible key hashes. Using virtual nodes somewhat helps to solve non-uniform key hash distribution, but it is not guaranteed; Also, data replication can help cache node faults.

Once the hash size for each server is fixed, it never changes even though they may have serious imbalances. Moreover, adding a new server may not significantly improve performance since node allocation is determined by hash values, which is a random allocation. Even worse, the consistent hashing scheme has no knowledge about the workload, which is a highly important variant [3].

As a motivating example, we randomly select 20,000 web pages among 1,106,534 pages in Wikipedia wikibooks database to profile key and value statistics. Figure 2(a) shows the number of objects when using 100 cache servers. Even though the hash function tends to provide uniformity, depending on the workloads the number of objects in each server can largely vary. The cache server that has the largest number of objects (659) has 15× more objects than the cache server with the smallest number of objects (42). This means that some cache servers use a lot more memory than others, which in turn worsens the performance. Figure 2(b) illustrates the object size has a large variation, potentially resulting in irregular hit rate to each server. Figure 2(c) describes the comparison between the number of objects and the size of objects in total. The two factors do not linearly increase so that it makes harder to manage the multiple number of servers. Figure 2(d) shows the average cache size per each object by dividing the total used cache size with the number of objects. From these statistics, we can easily conclude that consistent hashing needs to be improved with the knowledge of workloads.

3 System Design

The main argument against consistent hashing is that it can become very inefficient if the hash space does not represent the access patterns and cannot change over time to adapt to the current workload. The main idea of system design is that we adaptively schedule the hash space size for each memory cache server so that the overall performance over time improves. This is essential because currently once the size of hash space for each memory cache server is set, it never changes the configuration unless a new cache server is added or the existing server is deleted. However, adding/deleting a server does not have much impact since the assigned location is chosen randomly ignoring workload characteristics. Our system has three important phases: initial hash space assignment using virtual nodes, space partitioning, and memory cache server addition/removal. We first explain the memory cache architecture and assumptions used in the system design.

3.1 System Operation and Assumptions

There exist three extreme ways to construct a memory caching tier depending on the location of load-balancers: centralized architecture, distributed architecture, and hierarchically distributed architecture as shown in Figure 3. The centralized architecture handles all the requests from applications so that it can control hash space in one place which means object distribution can be controlled easily, whereas the load-balancers in the distributed architectures can have different configurations so that managing object distribution is hard. Since the centralized architecture is widely used structure in real memory caching deployments, we use this architecture in this paper. As load-balancers are implemented in a very efficient way minimizing the processing time, we

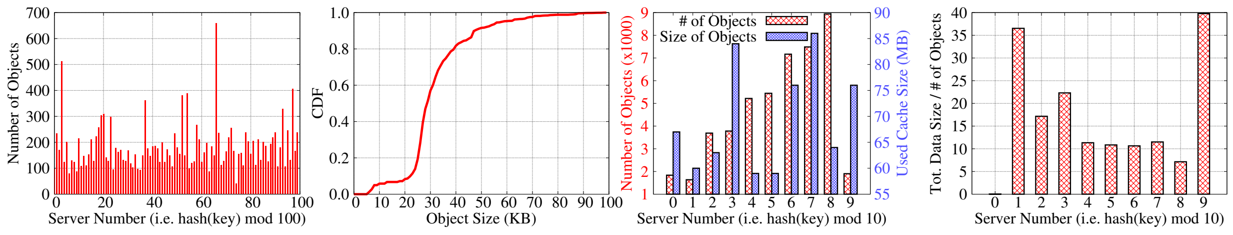

(a) Number of Objects (b) Object Size Distribution (c) # of Objects vs. Used Cache Size (d) Avg. Cache Size per Object

Figure 2: Wikibooks object statistics shows the number of objects in each server and used cache size are not uniform so that cache server performance is not optimized.

assume that the load-balancer does not become the bottleneck.

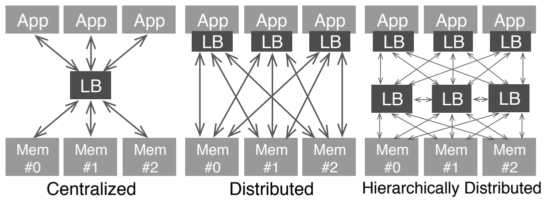


Figure 3: Memory Cache System Architecture; LB is load-balancer or proxy.

When a user requests a page from a web server application, the application sends one or more cache requests to a load-balancer – applications do not know there is a load-balancer since the implementation of the load-balancer is transparent. The load-balancer hashes the key to find the location where the corresponding data is stored, and sends the request to one of the memory cache servers (*get* operation). If there is data already cached, the data is delivered to the application and then to the user. Otherwise, the memory cache server notifies the application that there is no data stored yet. Then, the application queries the source medium such as database or file system to read the data, then sends it to the user and stores in the cache memory (*set* operation). Next time another user wants to read the same web site, the data is read from the memory cache server, resulting in a faster response time.

3.2 Initial Assignment

Consistent hashing mechanism can use "virtual nodes" in order to balance out the hash space over multiple memory cache servers so that different small chunks of the hash space can be assigned to each cache server. The number of virtual nodes is an administrative decision based on engineering experience, but it has no guarantee on the key distribution. Since our goal is to dynamically schedule the size of each cache server, we make a minimum bound on how many virtual nodes we need for schedulability.

Let $S = \{s_1, ..., s_{n_0}\}$ be a set of memory cache servers (the terms, memory cache server and node, are exchangeably used), where n_0 is the initial number of nodes. We denote v as the number of virtual nodes that each node has in the hash space H, and v_i as a virtual node i. That is, a node i can have $|s_i| = \frac{|H|}{n_0}$ objects, and a virtual node i can have $|v_i| = \frac{|H|}{n_0 \times v}$ objects, where $|H|$ is the total number of possible hash value. One virtual node can affect the other cache server in a clockwise direction as shown in Figure 1.

The key insight in our system is that in order to enable efficient repartitioning of the hash space, it is essential to ensure that each node has some region of the total hash space that is adjacent to every other node in the system. This guarantees that, for example, the most overloaded node has some portion of its hash space that is adjacent to the least loaded node, allowing a simple transfer of load between them by adjusting just one boundary. In order to allow every pair to influence each other, we need to make at least

$$v \geq \frac{{}^{n_0}\mathrm{P}_2}{n_0} = n_0 - 1, \qquad (1)$$

virtual nodes, where P is a permutation operation. Equation (1) guarantees that every node pair appears twice in a reverse order. So each physical node becomes $(n_0 - 1)$ virtual nodes, and the total number of overall virtual nodes becomes $n_0 \times (n_0 - 1)$. Also, we can increase the total number of virtual nodes by multiplying a constant to the total number. Figure 4 depicts an example assignment when there are five nodes. In a clockwise direction, every node influences all the other nodes.

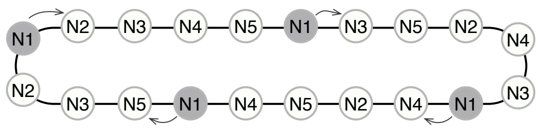


Figure 4: Assignment of Five Memory Cache Servers in Ring; As the example shows, N1 can influence all the other nodes N2, N3, N4, and N5. This applies to all the nodes.

Our node assignment algorithm is as follows. Let each node s_i have an array $s_{i,j} = \{(x,y) \mid 1 \leq x \leq n_0 \text{ and } y \in \{0,1\}\}$, where $1 \leq i \leq n_0$ and $1 \leq j \leq (n_0 - 1)$. Let $s^x_{i,j}$ and $s^y_{i,j}$ be x and y values of $s_{i,j}$, respectively. $s^x_{i,j}$ is defined as

$$s^x_{i,j} = \begin{cases} j & \text{if } j < i \\ j+1 & \text{if } j \geq i, \end{cases}$$

and all $s^y_{i,j}$ are initialized to 0. We pick two arbitrary numbers w_1 and w_2, where $1 \leq w_1 \leq n_0$ and $1 \leq w_2 \leq n_0 - 1$, assign w_1 in the ring, and label it as virtual node v_* in sequence ($*$ increases from 1 to $n_0 \times (n_0 - 1)$). Set $s^y_{w_1,w_2} = 1$, and $w_3 = s^x_{w_1,w_2}$. We denote $w_4 = (w_2 + k) \bmod n_0$, where $1 \leq k \leq n_0 - 1$. We then increment k from 1 to $n_0 - 1$, and check entries satisfying $s^y_{w_3,w_4} = 0$, and assign w_3 to w_1, w_4 to w_2, and $s^x_{w_3,w_4}$ to w_3. Repeat this routine until the number of nodes reaches $n_0 \times (n_0 - 1)$. For performance analysis, the time complexity of the assignment algorithm is $O(n_0^3)$ because we have to find $s^y_{w_3,w_4} = 0$ to obtain one entry each time, and there are $n_0 \times (n_0 - 1)$ virtual nodes. Therefore, the total time is $n_0(n_0 - 1)^2$. Note that this cost only needs to be paid once at system setup.

3.3 Hash Space Scheduling

As seen in Figure 2, key hash and object size are not uniformly distributed so that the number of objects and the size of used memory are significantly different, which in turn gives different performance for each memory cache server. The goal to use memory cache servers is to speed up response time to users by using a faster medium than the original source storage. Therefore, the performance of memory cache servers can be represented by the hit rate with the assumption that response time for all the cache servers are the same. However, usage ratio of each server should also be considered because the infrequent use of a cache server usually means the memory space is not fully utilized.

We define t_0 as the unit time slot for memory cache scheduling, which means the load-balancer repartitions the cache every t_0 time units. t_0 is an administrative preference that can be determined based on workload traffic patterns. Typically, only a relatively small portion of the hash space controllable by a second system parameter is rearranged during each scheduling event. If workloads are expected to change on an hourly basis, setting t_0 on the order of minutes will typically suffice. For slower changing workloads t_0 can be set to an hour.

In the load-balancer which distributes cache requests to memory cache servers, we can infer the cache hit rate based on the standard operations: *set* and *get*. A hit rate of a node s_i is

$$h_i = 1 - \frac{set(i)}{get(i) - set(i)},$$

where if $h_i > 1$, $h_i = 1$, and if $h_i < 0$, $h_i = 0$. “Hit rate” is a composite metric to represent both object sizes and key distribution, and this also applies when servers have different cache size. A simplified weighted moving average (WMA) with the scheduling time t_0 is used to estimate the hit rate smoothly over the scheduling times. Therefore, $h_i(t) = h_i(t-1)/t_0 + (1 - set(i)/get(i))$, where t is the current scheduling time and $t-1$ is the previous scheduling time. In each scheduling, $set(i)$ and $get(i)$ are reset to 0. We can also measure the usage ratio meaning how many requests are served in a certain period of time. The usage of a node s_i is $u_i = set(i) + get(i)$, and the usage ratio is $r_i = u_i / \max_{1 \leq j \leq n}\{u_j\}$, where n is the current number of memory cache servers, The usage ratio also uses a simplified WMA so that $r_i(t) = r_i(t)/t_0 + u_i / \max_{a \leq j \leq n}\{u_j\}$. In order to build up a scheduling objective with the hit rate and the usage ratio, we define a composite cost from hit rate and usage rate as $c = \alpha \cdot \overline{h} + (1-\alpha) \cdot r$, where $\alpha \in [0,1]$ is the impact factor to control which feature is more important and $\overline{h} = 1 - h$ is a miss rate, and state the scheduling objective as follows:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} (\alpha \cdot \overline{h_i} + (1-\alpha) \cdot r_i) \\ \text{subject to} & \overline{h_i} \in [0,1] \text{ and } r_i \in [0,1], \quad 1 \leq i \leq n \\ & \alpha \in [0,1] \end{array}$$

where n is the current number of the memory cache servers, and the objective is the sum of cost, and the conditions bind the normalized terms. This remains in a linear programming because we do not expand this to an infinite time span, which means the current scheduling state information propagates to the next scheduling only through hit rate WMA and usage ratio WMA. That is, we do not target to optimize all future schedulings, but the current workload pattern with small impact from past workload patterns. To satisfy the objective, we define the simple heuristic that finds the most cost disparity node pair with

$$s^*_{i,j} = max_{1 \leq i,j \leq n}\{c_i - c_j\}. \tag{2}$$

For performance analysis, since c is always nonnegative, this problem becomes the problem finding a maximum cost and a minimum cost. Therefore, we can find proper pairs in $O(n)$ because only neighbor nodes are considered to be compared. Equation (2) outputs a node pair where $c_i > c_j$, so a part of the hash space in c_i needs to move to c_j for balancing out. The load-balancer can either just change the hash space or migrate objects from c_i to c_j. Changing just hash space would provide more performance degradation than data migration because the old cache space in c_i should be filled in c_j again by reading slow data source medium. The amount of hash space is determined by the ratio of two nodes as c_j/c_i to balance the space. Also, we define $\beta \in (0,1]$ to control the amount of hash space moved from one node to the other node. Therefore, we move data from node s_i in a counter clockwise direction (i.e., decreasing direction) of the consistent hash ring for the amount of

$$\beta \cdot (1 - \frac{c_j}{c_i}) \times |s_i|. \tag{3}$$

For example, if we start with five inital memory cache servers, and at the first scheduling point with $c_i = 1$, $c_j = 0.5$ and $\beta = 0.01$ (1%), we have to move c_i with the amount of $0.01 \cdot (1 - \frac{0.5}{1}) \times \frac{2^{32}}{20} = 1,073,741$. This means 0.5% of the hash space from s_i moves to s_j. With traditional consistent hashing, there is no guarantee that s_i has hash space adjacent to s_j, but our initial hash assignment does guarantee all pairs of nodes have one adjacent region, allowing this shift to be performed easily without further dividing the hash space.

3.4 Node Addition/Removal

Most current memory cache deployments are fairly static except for periodic node failures and replacements. We believe that these cache deployments can be made more efficient by automatically scaling them along with workloads. Current cloud platforms allow virtual machines to be easily launched based on a variety of critiera, for example by using EC2's *as-create-launch-config* command along with its CloudWatch monitoring infrastructure [5].

The main goal of adding a new server is to balance out the requests across replicas that overall performance improves. Existing solutions based on consistent hashing rely on randomness to balance the hash space. However, this can not guarantee that a new server will take over the portion of the hash space that is currently overloaded. Instead, our system tries to more actively assign a balanced hash space to the new server. The base idea is that when servers are overloaded – the loads cross upward the threshold line defined in advance based on service level agreement (SLA) and sustain the overloaded states for a predefined period of time – we find the most overloaded k servers with $s_i^* = max^k_{1 \le i \le n}\{c_i\}$ and support them with new servers, where an operator max^k denotes finding top k values. So, n_0 number of virtual nodes are added as neighbors of s_i^*'s virtual nodes in the counter clockwise direction. The new server takes over exactly half of the hash space from s_i^*, which is $\frac{|s_i|}{2}$. The left part of Figure 5 illustrates that s_j is overloaded and s_k is added. s_k takes over a half of the hash space s_j has.

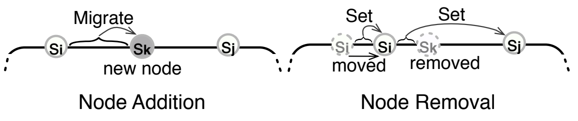


Figure 5: Object Affiliation in Ring After Node Addition and Removal

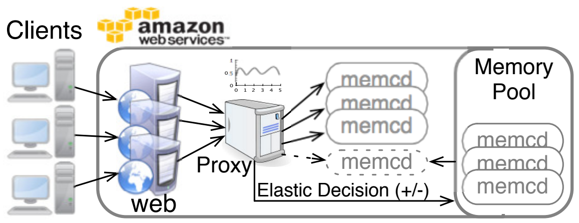
When a server is removed, the load-balancer knows about the removal by losing a connection to the server or missing keep-alive messages. Existing systems deal with node removal by shifting all the objects belonging to the removed node to the next node in a clockwise direction. However, this operation may make the next node overloaded and also misses a chance to balance the data over all the cache servers. When a node is removed in our system due to failure or managed removal – as with the adding criteria, the loads cross downward the threshold line and sustain the states – the two immediately adjacent virtual nodes will divide the hash space of the removed node. As shown in the right part of Figure 5, when there are three nodes s_i, s_k, and s_j in a clockwise sequence, and s_k is suddenly removed due to some reasons, the load-balancer decides how much hash space s_i moves based on the current costs c_i and c_j. s_i needs to move $\frac{c_j}{c_i+c_j} \times |s_j|$ amount of the hash space in a clockwise direction.

Of course, after a node is added or removed, the hash space scheduling algorithm will continue to periodically repartition hash space to keep the servers balanced.

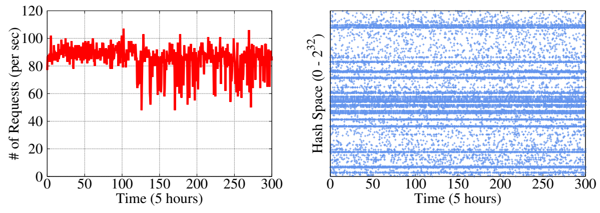
3.5 Implementation Considerations

To end our discussion of the system design, it is worth highlighting some of the practical issues involved in implementing the system in a cloud infrastructure. The scheduling algorithm is simple, and so is reasonable for implementation; however there exist two crucial aspects that must be addressed to deploy the system in the real infrastructure.

Data migration: When the scheduling algorithm schedules the hash space, it inevitably has to migrate some data from one server to another. Even though data are not migrated, the corresponding data are naturally filled in the moved hash space. However, since a response time between an original data source and a memory cache server are significantly different, users may feel slow response time [9]. The best way is to migrate the affected data behind the scene when the scheduling decision is made. The load-balancer can control the data migration by getting the data from the previous server and setting the data to the new server. The implementation should only involve the load-balancer since memory cache applications like memcached are already used in

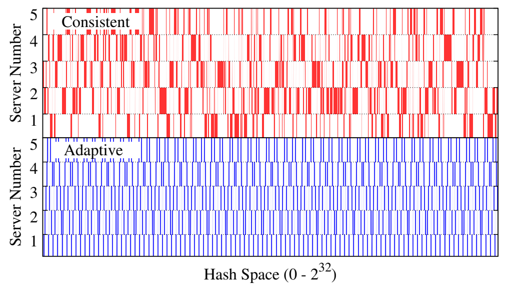


(a) Amazon EC2 Deployment

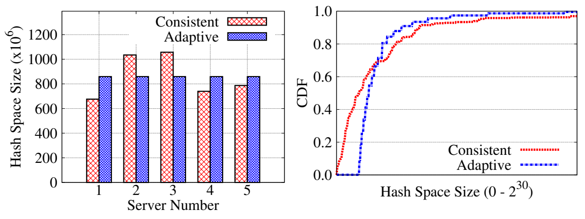


(b) Wikipedia Workload Characteristics: (1) # of Web Requests Per Second (left); (2) Key Distribution

Figure 6: Experimental Setup



(a) Initial Assignment Hash Map (5 servers)



(b) Hash Space Size (5 servers) (c) Hash Space Dist. (20 servers)

Figure 7: Initial hash space assignment with 5 - 20 memory cache servers.

many production applications. Also, Couchbase [6], an open source project, currently uses a data migration so that it is already publicly available.

Scheduling cost estimation: In the scheduling algorithm, the cost function uses the hit rate and the usage ratio because applications or load-balancers do not know any information (memory size, number of CPUs, and so on) about the attached memory cache servers. Estimating the exact performance of each cache server is challenging, especially under the current memory cache system. However, using the hit rate and the usage ratio makes sense because these two factors can represent the current cache server performance. Therefore, we implement the system as practical as possible to be deployed without any modifications to the existing systems.

4 Experimental Evaluation

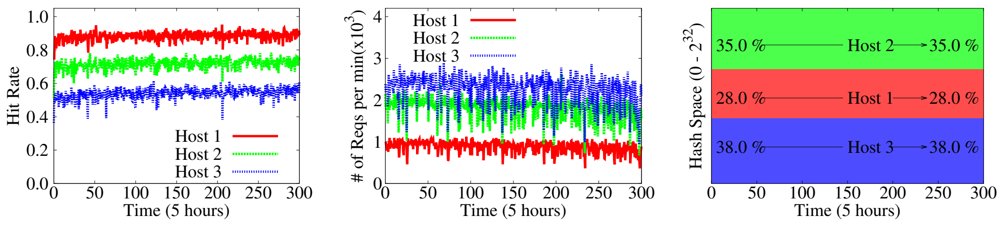
Our goal is to perform experiments in a laboratory environment to find out the scheduler behavior, and in a real cloud infrastructure to see the application performance. We use the Amazon EC2 infrastructure to deploy our system.

4.1 Experimental Setup

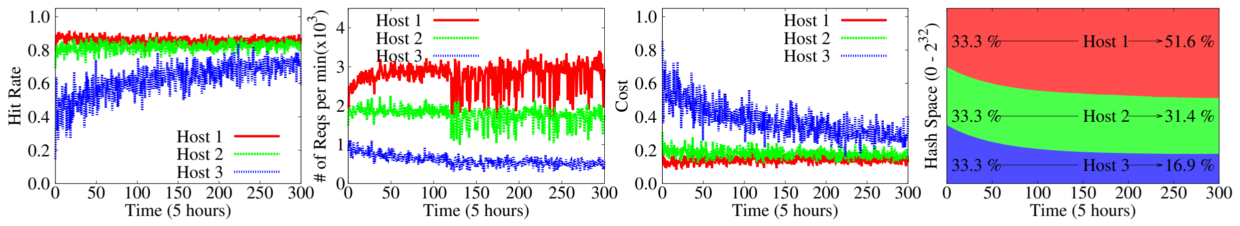
Laboratory System Setup: Five experimental servers, each of which has 4$\times$ Intel Xeon X3450 2.67GHz processor, 16GB memory, and a 500GB 7200RPM hard drive. Dom-0 is deployed with Xen 4.1.2 and Linux kernel 3.5.0-17-generic, and the VMs use Linux kernel 3.3.1. A Wikipedia workload generator, a web server, a proxy server, and memory cache servers are deployed in a virtualized environment. We use MediaWiki 1.14.1 [13], moxi 1.8.1 [15], and memcached 1.4.15 [14]. MediaWiki has global variables to specify whether it needs to use memory cache: wgMainCacheType, wgParserCacheType, wgMessageCacheType, wgMemCachedServers, wgSessionsInMemcached, and wgRevisionCacheExpiry. In order to cache all texts, we need to set wgRevisionCacheExpiry with expiration time, otherwise MediaWiki always retrieves text data from database.

Amazon EC2 System Setup: As shown in Figure 6(a), web servers, proxy, and memory cache servers are deployed in Amazon EC2 with m1.medium – 2 ECUs, 1 core, and 3.7 GB memory. All virtual machines are in us-east-*. Wikipedia clients are reused from our laboratory servers.

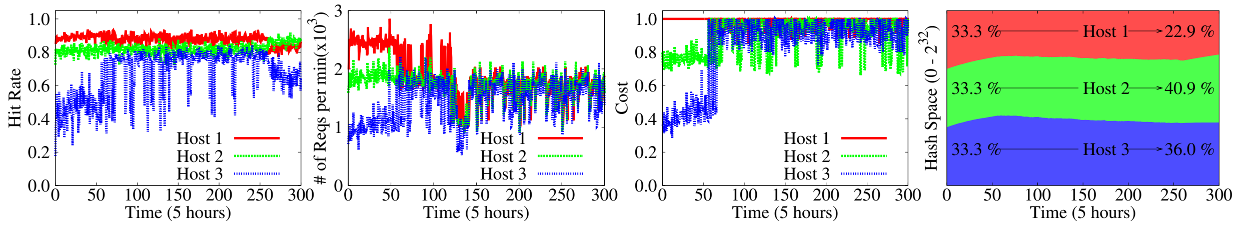
Wikipedia Trace Characteristics: Wikipedia database dumps and request traces have been released to support research activities [21]. January 2008 database dump and request traces are used in this paper. Figure 6(b) shows the trace characteristics of Wikipedia after we have scaled down the logs through sampling. Figure 6(b)(1) illustrates the number of requests per second from a client side to a Wikipedia web server. Requests are sent to a web server, which creates the requests sent to a proxy server and to individual memory cache servers depending on the hash key. Figure 6(b)(2) depicts the key distribution over the hash space 2^{32} range – most keys are the URL without a root address (e.g., ~~http://en.wikipedia.org/~~wiki/3D_Movie).



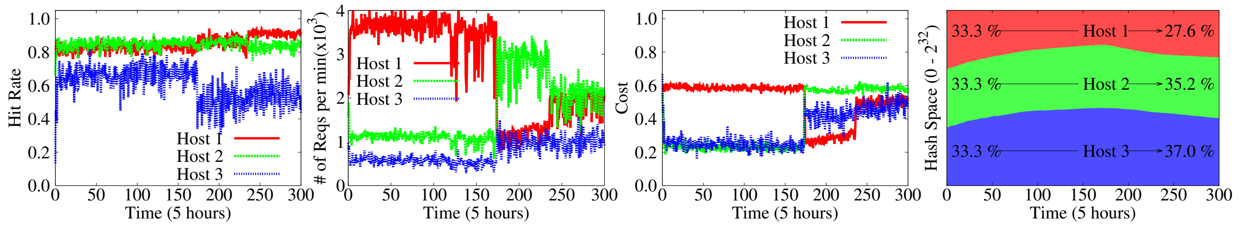
(a) Consistent Hashing



(b) Adaptive Hashing ($\alpha = 1.0$ and $\beta = 0.01$)



(c) Adaptive Hashing ($\alpha = 0.0$ and $\beta = 0.01$)



(d) Adaptive Hashing ($\alpha = 0.5$ and $\beta = 0.01$)

Figure 8: Hash Space Scheduling with Different Scheduling Impact Values α and β.

4.2 Initial Assignment

As we explain in Section 2 and 3.2, the initial hash space scheduling is important. Firstly, we compare the hash space allocation with the current system, ketama [11] – an implementation of consistent hashing. Figure 7 illustrates the initial hash space assignment. Figure 7(a) shows the difference between the consistent hashing allocation and adaptive hasing allocation when there are five memory cache servers. The number of virtual nodes is 100 (system default) for the consistent hashing scheme so that the total number of virtual nodes is 5×100. Our system uses the same number of virtual nodes by increasing the number of virtual nodes per physical node by a factor of $\frac{100}{(n_0-1)}$. With $n_0 = 5$, the total number of virtual nodes in our system is $5 \times 4 \times \frac{100}{4} = 500$. Consistent hashing has an uneven allocation without knowledge of workloads, which is bad. Adaptive hasing starts with the same size of hash space to all the servers, which is fair. Figure 7(b) compares the size of hash space allocated per node with each technique. In consistent hashing, the largest gap between the biggest hash size and the smallest hash size is 381,114,554. This gap can make a huge performance difference between memory servers. Figure 7(c) shows the hash size distribution across 20 servers—our approach has a less variability in the hash space assigned per node. Even worse, the consistent hashing allocation fixes the assignment based on a server's address, and does not adapt if the servers are not utilized well. We can easily see that without knowledge of workloads, it is hard to manage this allocation to make all the servers perform well in a balanced manner.

4.3 α Behavior

As described in Section 3.3, we have two parameters α and β to control the behaviors of the hash space scheduler. α gauges the importance of hit rate or usage rate. $\alpha = 1$ means that we only consider the hit rate as a metric of scheduling cost. $\alpha = 0$ means that we only consider the usage rate as a metric of scheduling cost. β is the ratio of the hash space size moved from one memory server to another. Since β changes the total scheduling time and determines fluctuation of the results, we fix β as a 0.01 (1%) based on our experience running many times. In this experiment, we want to see the impact of α parameter. Particularly, we check how α changes hit rate, usage rate, and hash space size. Our default scheduling frequency is 1 min.

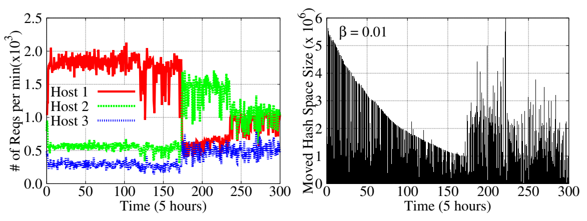
As a reference, Figure 8(a) illustrates how the current consistent hashing system works under the Wikipedia workload. The default hash partitioning leaves the three servers unbalanced, causing significant differences in the hit rate and utilization of each server.

Figure 8(b) shows the performance changes when $\alpha = 1.0$, which means we only consider balancing hit rates among servers. As Host 3 starts with a lower hit rate than other two hosts, the hash scheduler takes a hash space from Host 3 in order to increase its hit rate. The usage rate of host 3 decreases as its hash allocation decreases.

Figure 8(c) depicts the results when $\alpha = 0.0$, which means we only seek to balance usage rates among servers. The system begins with an equal hash space allocation across each host, but due to variation in object popularity, each host receives a different workload intensity. Since Host 1 initially has about 2.5 times as many requests per second arriving to it, the scheduler tries to rebalance the load towards Hosts 2 and 3. The system gradually shifts data to these servers, eventually balancing the load so that the request rate standard deviation across servers drops from 0.77 to 0.09 over the course of the experiment. This can be seen from the last (fourth) figure in Figure 8(c).

To balance these extremes, we next consider how α value (0.5) affects the performance. Figure 8(d) shows hit rate and usage rate of each server with $\alpha = 0.5$. Since the cost of each server is calculated out of hit rate and usage rate, the scheduler tries to balance both of them. As shown in the third graph in Figure 8(d), the costs balance among three servers which also means balancing both hit rate and usage rate.

Since workloads have different characteristics, the parameters α and β should be adjusted accordingly. We show further aspects of this adjustment while experimenting the system in the Amazon EC2 infrastructure.



(a) # Requests of Each Host (b) Moved Hash Size in Each Scheduling Time

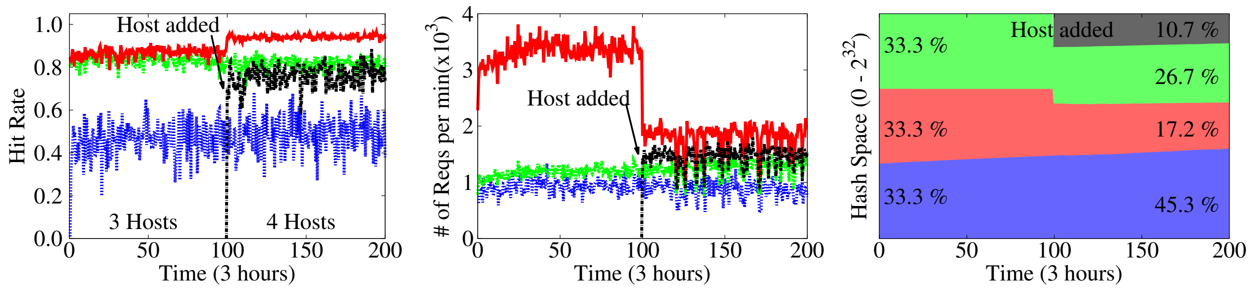
Figure 10: Hash Space Scheduling Analysis

4.4 β Behavior

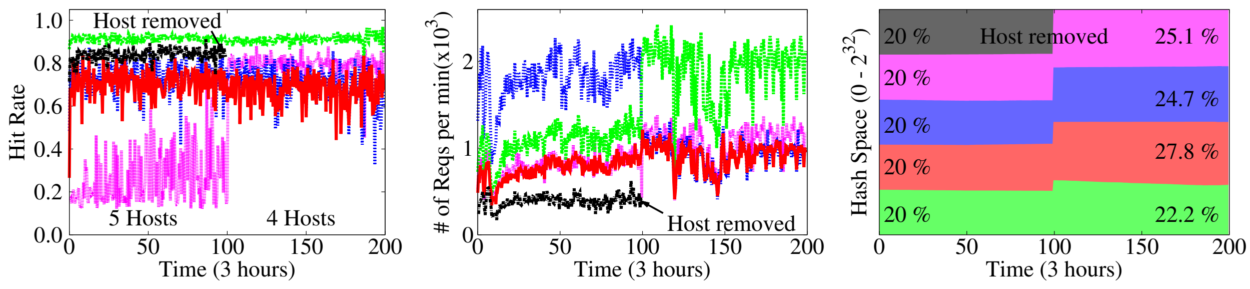
β value is the ratio of the amount of hash size moved in each scheduling time. We can show the behavior of β by illustrating the number of requests from each server (Figure 10(a)) and the amount of hash size per scheduling time (Figure 10(b)). As β value 0.01 yields approximately 1% of hash space from Equation (3), the moved hash size is decreasing as the hash space of an overloaded server decreases. Figure 10(b) shows the amount of hash space moved each interval. This continues to fall as Host 1's hash space decreases, but has relatively little effect on the request rate. However, after 180 minutes, a small, but very popular region of the hash space is shifted to Host 2. The system responds by trying to move a larger amount of data between hosts in order to rebalance them. This illustrates how the system can automatically manage cache partitioning, despite highly variable workloads.

4.5 Scaling Up and Down

As we explained in Section 3.4, adaptive hasing can autonomously add or delete memory cache servers based on the current performance. Since cloud infrastructure hosting companies provide a function to control the resources elastically, this is a very useful feature to prevent performance issues due to traffic bursts situation. Figure 9 shows the performance impact when adding a new server or deleting a server from the memory cache tier. Figure 9(a) starts with three memory cache servers, and a new server is added at 100 minutes due to an overloaded server. When a new server is added, the overloaded server gives 30% of its traffic to the new server so that overall usage rates of all servers are balanced. Conversely, Figure 9(b) assumes that one server out of five servers crashes at 100 minutes. As our initial hash allocation assigns the servers adjacent to one another, this gives a good benefit by distributing hash space to all other servers. This can be seen from the third graph in Figure 9(b).



(a) Node Addition



(b) Node Deletion

Figure 9: Memory Cache Node Addition / Deletion ($\alpha = 0.5$ and $\beta = 0.01$).

4.6 User Performance Improvement

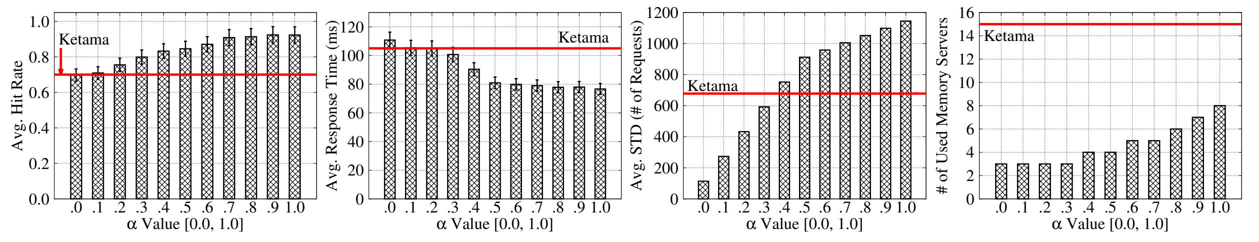
The previous experiments have demonstrated how the parameters affect the adaptive hash scheduling system; next we evaluate the overall performance and efficiency improvements it can provide. We use Amazon EC2 to run up to twenty total virtual machines — three web servers, one proxy server, one database, and between 3 and 15 memory cache servers. We use five servers in our own lab to act as clients, and have them connect to the web servers using the Wikipedia workload.

We compare two caching approaches: a fixed size cluster of fifteen caches partitioned using Ketama's consistent hashing algorithm and our adaptive approach. The workload starts at 30K req/min, rises to 140K req/min, and then falls back to the original load over the course of five hours, as shown in Figure 12. We configure Ketama for a "best case scenario"—it is well provisioned and receives an average response time of 105 ms, and a hit rate of 70%. We measure our system with α values between 0 and 1, and initially allocate only 3 servers to the memcached cluster. Our system monitors when the request rate of a cache server surpasses a threshold for more than 30 seconds to decide whether to add or remove a node from the memory server pool. For this experiment, we found that more than 6K requests/sec caused a significant increase for the response time, so we use this as the threshold.

Figure 11 shows (a) average hit rate; (b) average response time from clients; (c) average standard deviation on number of requests to the EC2 cache servers; (d) number of used servers including dynamically added ones. Horizontal lines show the average performance of the current consistent hashing system used by moxi, and bars represent our system with different α values.

As expected, increasing α causes the hit rate to improve, providing as much as a 31% increase over Ketama. The higher hit rate can lower response time by up to 38% (figure b), but this is also because a larger α value tends to result in more servers being used (figure d). Since a large α ignores balance between servers (figure c), there is a greater likelihood of a server becoming overloaded when the workload shifts. As a result, using a high α does improve performance, but it will come at increased monetary cost for using more cloud resources. We find that for this workload, the system administrators may want to assign $\alpha = 0.5$, which achieves a reasonable average response time while requiring only a small number servers compared to Ketama.

Figure 12 shows how the workload and number of active servers changes over time for $\alpha = 1$. As the workload rises, the system adapts by adding up to five additional servers. While EC2 charges in full hour increments, our system currently aggressively removes servers when they are no longer needed; this behavior could easily be changed to have the system only remove servers at the end of each instance hour.



(a) Average Hit Rate (b) Average Response Time (c) Average STD on # of Requests (d) # of Used Memory Servers

Figure 11: Amazon EC2 Deployment: Five Workload Generators, Three Web Servers, One Database, and Total 15 Memory Cache Servers in Memory Cache Pool; Three memory cache servers are used initially.

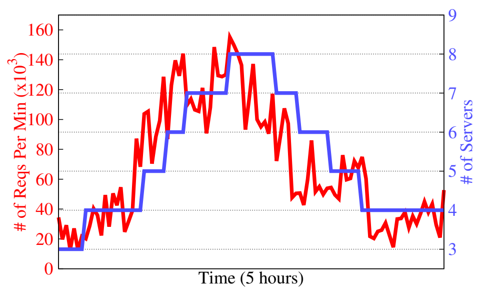


Figure 12: Number of cache servers adapting dynamically based on the workload intensity.

5 Related Work

Peer-to-peer applications gave rise to the need for distributed lookup systems to allow users to find content across a broad range of nodes. The Chord system used consistent hashing algorithms to build a distributed hash table that allowed fast lookup and efficient node removal and addition [20]. This idea has since been used in a wide range of distributed key-value stores such as memcached [14], couchbase [6], FAWN [2], and SILT [12]. Rather than proposing a new key-value store architecture, our work seeks to enhance memcached with adaptive partitioning and automated replica management. Previously, memcached has been optimized for large scale deployments by Facebook [16, 18], however their focus is on reducing overheads in the network path, rather than on load balancing. Zhu et. al. [22] demonstrate how scaling down the number of cache servers during low load can provide substantial cost savings, which motivates our efforts to build a cache management system that is more adaptable to membership changes. Christopher et. al. [19] proposes a prediction model to meet the strict service level objectives by scaling out using replication.

There are many other approaches for improving the performance of key-value stores. Systems built upon a wide range of hardware platforms have studied, including low-power servers [2], many-core processors [4], having front-end cache [8], and as combined memory and SSD caches [17]. While our prototype is built around memcached, which stores volatile data in RAM, we believe that our partitioning and replica management algorithms could be applied to a wide range of key-value stores on diverse hardware platforms.

Centrifuge [1] proposes a leasing and partioning model to provide the benefits of fine-grained leases to in-memory server pools without their associated scalability costs. However, the main goal of Centrifuge is to provide a simplicity to general developers who can use the provided libraries to model leasing and partioning resources. This work can be applied to managing the memory cache system, but Centrifuge does not support dynamic adaptation to workloads.

6 Conclusion

Many web applications can improve their performance by using distributed in-memory caches like memcached. However, existing services do not provide autonomous adjustment based on the performance of each cache server, often causing some servers to see unbalanced workloads. In this paper we present how the hash space can be dynamically re-partitioned depending on the performance. By carefully distributing the hash space across each server, we can more effectively balance the system by directly shifting load from the most to least loaded servers. Our adaptive hash space scheduler balances both the hit rate and usage rate of each cache server, and the controller can decide automatically how many memory cache servers are required to meet the predefined performance. The partitioning algorithm uses these parameters to dynamically adjust the hash space so that we can balance the loads across multiple cache servers. We implement our system by extending memcached and an open source proxy server, and test both in the lab and in Amazon EC2. Our future works include an automatic α value adjustment according to the workloads and a micro management of hot objects without impacting application performance.

Acknowledgments: We are grateful to our reviewers and shepherd for their comments. This work was supported in part by NSF grant CNS-1253575.

References

[1] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Amazon CloudWatch. `http://aws.amazon.com/cloudwatch`.

[6] Couchbase. vbuckets: The core enabling mechanism for couchbase server data distribution. *Technical Report*, 2013.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[8] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[9] Adam Wolfe Gordon and Paul Lu. Low-latency caching for cloud-based web applications. *NetDB*, 2011.

[10] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Netw.*, 31(11-16):1203–1213, May 1999.

[11] Ketama. `http://www.audioscrobbler.net/development/ketama/`. 2013.

[12] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[13] MediaWiki. `http://www.mediawiki.org/wiki/MediaWiki`.

[14] Memcached. http://memcached.org.

[15] Moxi. http://code.google.com/p/moxi.

[16] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcached at facebook. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[17] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. *2012 41st International Conference on Parallel Processing*, 0:470–479, 2012.

[18] Paul Saab. Scaling memcached at facebook, `http://www.facebook.com/note.php?note\_id=39391378919`.

[19] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. *Internation Conference on Autonomic Computing*, 2013.

[20] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[21] Erik-Jan van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master Thesis*, 2009.

[22] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

Exploiting Processor Heterogeneity for Interactive Services

Shaolei Ren[‡*] Yuxiong He[†] Sameh Elnikety[†] Kathryn S. McKinley[†]
[‡]*Florida International University* [†]*Microsoft Research*

Abstract

To add processing power under power constraints, emerging heterogeneous processors include fast and slow cores on the same chip. This paper demonstrates that this heterogeneity is well suited to interactive data center workloads (e.g., web search, online gaming, and financial trading) by observing and exploiting two workload properties. (1) These workloads may trade response quality for responsiveness. (2) The request service demand is unknown and varies widely with both short and long requests. Subject to per-server power constraints, traditional homogeneous processors either include a few high-power fast cores that deliver high quality responses or many low-power slow cores that deliver high throughput, but not both.

This paper shows heterogeneous processors deliver both high quality and throughput by executing short requests on slow cores and long requests on fast cores with Fast Old and First (FOF), a new scheduling algorithm. FOF schedules new requests *with unknown service demands* on the fastest idle core and migrates requests from slower to faster cores. We simulate and implement FOF. In simulations modeling Microsoft's Bing index search, FOF on heterogeneous processors improves response quality and increases throughput by up to 50% compared to homogeneous processors. We confirm simulation improvements with an implementation of an interactive finance server using Simultaneous Multithreading (SMT), configured as a dynamic heterogeneous processor. Both simulation and experimental results indicate processor heterogeneity offers a lot of potential for interactive workloads.

1 Introduction

A heterogeneous processor contains cores with differentiated power and performance characteristics. All cores execute the same instruction set (ISA), but they run at different speeds and/or use different microarchitectures so that the faster the core, the more power it consumes. Since power consumption increases faster than speed, a *fast* core executes a request in less time than a *slow* core, but consumes more energy. We show that a homogeneous processor under a fixed power constraint can only deliver either high performance with a few fast cores or high throughput with many more slow cores for interactive data center workloads, whereas heterogeneous processors deliver both with substantial benefits.

Interactive services require high quality results and timely responses to satisfy users and generate revenue [23]. For example, Bing search servers are provisioned to execute requests within 120 ms with an average quality of 0.99. This *quality* metric compares returned search results within a time limit to an off-line search with unlimited time and resources. Interactive services have two properties. First, many interactive services — including web search, financial trading, and online gaming — are *adaptive*, i.e., processing a request for more time improves response quality. Adaptive execution may return lower-quality responses (or partial results) for responsiveness. Second, the service demand of a request is unknown a priori, and it varies widely with both short and long requests.

The main contribution of this paper is to demonstrate that exploiting processor heterogeneity delivers substantial benefits to interactive workloads in data centers as compared to using homogeneous processors. More concretely, when building a data center to support interactive services, power budgets constrain the overall system as well as individual servers. Such design-time power constraints limit the core speed and number of cores on a chip. With a fixed design-time power budget, system designers can deploy a homogeneous processor with either fewer *fast* high-performance power-hungry cores that are less energy-efficient or a larger number of *slow* cores that are more energy-efficient. For example, one fast core may burn as much power as 8 to 16 low power cores [38]. Fast cores offer high service quality and fast response at a light load, but when the load increases, both quality and throughput degrade quickly since requests significantly outnumber cores. In contrast, more slow energy-efficient

*This work was partially done while Shaolei Ren was visiting Microsoft Research.

cores increase throughput by executing more requests, but the quality of responses drops when the slow cores do not execute fast enough to fully process long requests before their respective deadlines. We show how to achieve both high quality and throughput with a heterogeneous design provisioned with both fast and slow cores. The key idea is to execute short requests on slow cores, so that they complete within their deadline with low energy consumption, and to execute long requests on fast cores to obtain high response quality. Towards this end, there are two challenges. (1) When a request arrives, we do not know its service demand, and predicting service demand is difficult for many workloads [34]. (2) Even if we know the service demand, there are multiple requests but only a limited number of cores, and therefore the most appropriate core on which to execute a request may not be available.

This paper address these challenges by introducing a new online algorithm for scheduling requests of interactive services on heterogeneous processors, called Fast Old and First (FOF). When a new request arrives, FOF assigns it the fastest available core (*Fast First*). When a request completes, FOF promotes the oldest request on a slower core to the fastest, newly available core (*Fast Old*). FOF achieves high throughput because it completes many short requests on energy-efficient slow cores. FOF achieves high response quality since requests are processed on fast cores whenever possible and long requests execute with a higher probability on fast cores and thus all requests are likely to complete before their deadlines.

We model Microsoft Bing, a commercial web search engine. We measure Bing workload distributions and quality profiles. We find that the request service demand has high variance, and response quality is monotonically increasing in processing time (before the deadline). Our simulation results of web search show that FOF on heterogeneous processors achieves a significantly higher response quality and up to 50% improvement in throughput compared to homogeneous processors with the same design-time power budget as well as compared to traditional scheduling algorithms. We also show that FOF improves throughput and quality for a variety of heterogeneous hardware configurations and workloads with various service demand distributions. Moreover, FOF improves average quality, reduces quality variance, improves high-percentile quality, and improves throughput.

We further show how to configure a Simultaneous Multithreading (SMT) core as a dynamic heterogeneous processor and modify FOF for it. A core executing one thread acts as a fast core, while a core executing $M > 1$ SMT hardware threads acts as M slow cores. Even with the limited heterogeneity of a 2-way 6 core SMT processor, FOF improves the *measured performance* of an interactive finance server by up to 16% compared to default round-robin OS scheduling and by 27% when SMT is turned off. We validate our simulator against these measurements. FOF in implementation performs the same or better than in simulation. These results indicate that FOF offers performance benefits for heterogeneity present in data centers today.

We show how to compute the number of slow and fast cores in the hardware configurations using the request service demand distribution. In general, more long requests require more fast cores, and more slow cores are required for sustaining a higher throughput. Future data centers can exploit these results either to substantially reduce the number of servers, or to increase the capacity without compromising quality or responsiveness.

2 Scheduling Model

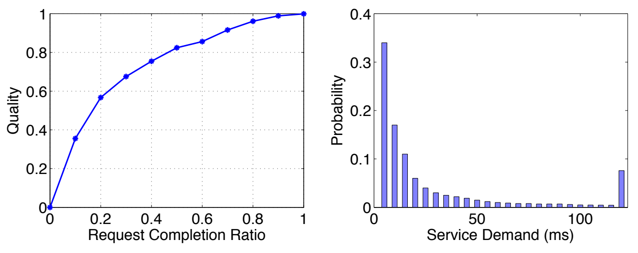
This section measures request characteristics in interactive services for a commercial web search engine, and formalizes our job and hardware model.

The literature establishes the following characteristics of interactive services [48, 35, 38, 26]. **(1) Adaptive execution.** Adaptive execution partially evaluates a request and returns a response before completion; more computation yields better quality. **(2) Concave quality profile.** If a request executes fully, it receives quality 1 (measured off-line). A concave function exhibits diminishing returns and captures the relationship between quality and computational resources (see Section 2.1). **(3) Deadline.** For online interactive services, users expect timely responses. Long response times are not acceptable because they cause user dissatisfaction and loss of revenue [23]. We express timing constraints as deadlines.

Recommendation systems, ad services, and video streaming have similar characteristics. For example, in scalable video coding, basic layers are more important than refinement layers, and the quality of received video streams improves monotonically with the number of layers but exhibits diminishing returns [28]. This paper focuses on web search and finance, but our results apply more broadly.

2.1 Measurement Study of Bing Search

We confirm these characteristics with measurements of Bing. Bing's web index serving system receives user queries and returns the response. This system is a distributed interactive search service. The web index contains billions of documents and thus, the index is partitioned and managed across thousands of servers. To meet responsiveness goals, Bing and other modern search engines [48, 35, 38] design and configure search such that the index for a server fits in main memory of the server with virtually no I/O or system calls, creating a CPU bound process. Other interactive services, such as video streaming, have similar goals and requirements. When a request arrives, the system assigns it to an aggregator, which sends the request to servers. Each server returns its matched results to the aggregator. The aggregator collects them and returns the top L webpages to the user.



(a) Quality profile. (b) Service demand distribution.

Figure 1: Measured workload of Bing search.

Search servers support adaptive execution with response deadlines. Each web search query contains a set of keywords. A server scans an inverted index looking for webpages that match the keywords and ranks the matching webpages. The response is links to the top documents that match the keywords. The more time the server spends in matching and ranking the documents, the better quality (i.e., more relevant) the search results. If the search server does not finish searching the entire index within the deadline, it returns the best matches so far. The server responds to the aggregator within 120 ms. The aggregator returns its collected results to users without waiting for any delayed responses. Ranking involves complex calculations and search servers are computationally intensive [38].

Quality Profile We obtained 200K queries from a production trace of Bing queries. We execute them multiple times with different completion deadlines using Bing in a controlled setting. Response quality compares the documents returned to the *golden* results (Quality = 1), without any deadline, such that Bing fully processes each request. The x-axis of Figure 1(a) is the *request time completion ratio* which is calculated as the actual measured processing time divided by its full processing time. The y-axis is the average response quality. Figure 1(a) shows that Web search has a monotonically increasing and (roughly) concave response quality profile.The concavity is because the inverted index searches popular webpages first, which are more likely to rank higher and contribute more to the response quality.

Service Demand Figure 1(b) presents the measured service demand distribution for 200K queries. Request service demand varies with many short requests, less than 40 ms, and under the 120 ms deadline, over 10% have a demand greater than or equal to 100 ms. This diversity in request service demands has been observed in many workloads [10, 48, 24, 25].

2.2 Job Model

An individual request is processed *sequentially*, while multiple requests can be processed on different cores *concurrently*. A job (request) is specified by a tuple (t_a, d, w), where t_a is arrival time, d is lifespan, and w is service demand. The job deadline is $t_a + d$. We assume d is the same for all jobs without loss of generality. We denote the maximum service demand by $\hat{w}$. The service demand is the total work (i.e., CPU cycles) required to complete a request and is unknown until the job completes. However, the service demand distribution is available by using online or offline profiling [34, 26, 25]. Thus, $w \in [0, \hat{w}]$ is an unknown random variable, whose probability density function (PDF) and cumulative distribution function (CDF) are denoted by $f(w)$ and $F(w) = \int_0^w f(x)dx$, respectively. We can alternatively use discrete values of service demands (e.g., measured values shown in Figure 1(b)) and all the analysis still applies, where PDF $f(w)$ is the probability mass function and w takes values from a finite set.

The server can fully process a request, returning a complete result, or terminate with a partial result. We measure the actual *quality* $q(r) : \mathbb{R}^+ \to \mathbb{R}$ off-line, comparing processed work to demand. While each request may have a unique quality profile that is unavailable to an online scheduler, we use $q(r)$, as shown in Figure 1(a), to represent expected quality profile of a request.

2.3 Hardware Model

Limited by power constraints, architects are turning to parallelism and heterogeneity in search of power-efficient performance [3, 31, 36, 8, 33]. A heterogeneous processor consists of $N > 1$ heterogeneous cores, indexed by $1, 2, \cdots, N$, which offer non-uniform performance and power consumption due to processing speeds or microarchitecture or both. Without loss of generality, we assume that the i-th core performance (i.e., effective speed) s_i and power p_i satisfy $0 < s_1 \le s_2 \le \cdots \le s_N$ and $0 < p_1 \le p_2 \le \cdots \le p_N$ [34]. Moreover, we assume that a core with higher performance speed burns more power to process a unit of work, i.e., forall $1 \le i \le j \le N$, $p_i/s_i \le p_j/s_j$, since otherwise the faster core is more energy-efficient than the slower core and there is no need to include the slower core in the processor design space. To fairly compare heterogeneous to homogeneous processors, we give them all the same design-time power budget.

3 Scheduling Algorithm

The scheduling objective of FOF is to improve the average response quality of all requests and thereby increase throughput. In practice, data center designers may exploit throughput improvements either by generating and servicing more load per server or by supporting the same load with fewer servers.

3.1 Key Insights

Intuitively, we want to schedule long requests on fast cores (since only fast cores are sufficient to ensure timely and high-quality responses for long requests) and schedule short requests on slow cores (since they are sufficient to ensure timely and high-quality responses). An ideal scheduler will thus maximize throughput and quality by executing every request on the slowest core that can meet the request deadline and quality requirement. However, there are two challenges. (1) *Assignment*: since the request service demand is unknown, how do we assign short requests to slow cores and long requests to fast cores? (2) *Availability*: given multiple requests and a limited number of cores, the most appropriate core may not always be available.

(1) Assignment FOF *migrates requests from slow to fast cores during its execution.* This policy increases the probability that short requests complete on slow cores and when a fast core becomes available, it processes longer requests. Given a deadline, this policy improves total response quality of requests, sustaining higher throughput while satisfying the target quality.

Theorem 1 explains why "slow to fast" improves throughput. The theorem formally establishes that migrating a request from slower to faster cores during its execution is the most energy-efficient schedule. Given the server's design-time power budget, dynamic energy per time unit is bounded. Thus, when each individual request consumes less energy, the server can serve more requests, improving throughput. Theorem 1 assumes *the desired core(s) are always available*, and later we address multiple requests competing for cores.

Theorem 1 *Given request deadline d, service demand CDF F, and a quality profile function q that is monotonically increasing and concave. To meet any average quality requirement, the core speed for processing the request is non-decreasing under an optimal schedule that minimizes the average CPU energy consumption of the request.*

Proof. We first meet the quality requirement. Request quality is a function of the quality profile q and work completed before the deadline. Let us define the target work $\bar{x}$, which specifies the maximum amount of work completed prior to the deadline regardless of the actual service demand. If the request has a total service demand less than $\bar{x}$, the request runs until completion, whereas the request is terminated at work $\bar{x}$ otherwise. Since the quality profile function q is monotonically increasing in $\bar{x} \in [0, \hat{w}]$, the expected average response quality increases from 0 to 1. Therefore, given an average quality requirement $0 \leq r \leq 1$, we can find a fixed target work $\bar{x} \in [0, \hat{w}]$ that satisfies the quality target. After finding the target work $\bar{x}$, we formulate the energy-minimization problem for scheduling a request as follows:

$$\min_{\mathscr{X}} \int_0^{\bar{x}} \left[1 - F(x)\right] \cdot \frac{p_{\mathscr{X}}(x)}{s_{\mathscr{X}}(x)} dx, \tag{1}$$

$$s.t., \qquad \int_0^{\bar{x}} \frac{1}{s_{\mathscr{X}}(x)} dx \leq d, \tag{2}$$

where $\mathscr{X}$ is a schedule that specifies the order and cores that process the single request, $s_{\mathscr{X}}(x) \in \{s_1, s_2 \cdots s_N\}$, and $p_{\mathscr{X}}(x) = p_i$, if $s_{\mathscr{X}}(x) = s_i$. Constraint (2) guarantees that the schedule $\mathscr{X}$ satisfies the deadline. We now prove the theorem by contradiction. Suppose that $s_{\mathscr{X}'}(x)$ minimizes (1) while, under the schedule $\mathscr{X}'$, the job is first processed by a faster core and then by a slower one. Thus, there exist x_1 and x_2 such that $0 \leq x_1 < x_1 + dx \leq x_2 < x_2 + dx \leq \bar{x}$ and $s_{\mathscr{X}'}(x_1') > s_{\mathscr{X}'}(x_2')$, where $x_1' \in [x_1, x_1 + dx]$, $x_2' \in [x_2, x_2 + dx]$ and dx is a sufficiently small positive number.

Since we assume faster cores consume more energy to process one unit of work than slower ones, the following inequality holds:

$$\begin{aligned} &[1 - F(x_1')] \cdot \left[\frac{p_{\mathscr{X}'}(x_1')}{s_{\mathscr{X}'}(x_1')} - \frac{p_{\mathscr{X}'}(x_2')}{p_{\mathscr{X}'}(x_2')}\right] \\ &+ [1 - F(x_2')] \cdot \left[\frac{p_{\mathscr{X}'}(x_2')}{s_{\mathscr{X}'}(x_2')} - \frac{p_{\mathscr{X}'}(x_1')}{s_{\mathscr{X}'}(x_1')}\right] \\ &= \left[\frac{p_{\mathscr{X}'}(x_1')}{s_{\mathscr{X}'}(x_1')} - \frac{p_{\mathscr{X}'}(x_2')}{s_{\mathscr{X}'}(x_2')}\right] \cdot [F(x_2') - F(x_1')] > 0. \end{aligned} \tag{3}$$

Thus, we have $[1 - F(x_1')] \cdot \frac{p_{\mathscr{X}'}(x_1')}{s_{\mathscr{X}'}(x_1')} + [1 - F(x_2')] \cdot \frac{p_{\mathscr{X}'}(x_2')}{s_{\mathscr{X}'}(x_2')} > [1 - F(x_1')] \cdot \frac{p_{\mathscr{X}'}(x_2')}{s_{\mathscr{X}'}(x_2')} + [1 - F(x_2')] \cdot \frac{p_{\mathscr{X}'}(x_1')}{s_{\mathscr{X}'}(x_1')}$. By evaluating the integral in (1), the expected energy consumption is further reduced by exchanging the order of cores processing the x_1'–th cycle and the x_2'–th cycle, for $x_1' \in [x_1, x_1 + dx]$ and $x_2' \in [x_2, x_2 + dx]$, while keeping the rest of the schedule $\mathscr{X}'$ unchanged. This contradicts the assumption that $\mathscr{X}'$ minimizes (1) and hence, proves Theorem 1. ■

The most obvious application of this theorem is optimizing dynamic run-time energy of service requests. However, we leave dynamic energy to future work and focus on design-time, because interactive service providers must *first* determine whether or not a heterogeneous processor can improve throughput and quality given design-time power constraints, and if it can, what combinations of fast and slow processors to use. We next leverage the insight of migrating a request from slower to faster cores to improve quality and throughput.

(2) Availability Given multiple requests and a limited number of cores, the most appropriate core to execute a request may not be available. FOF solves this problem by assigning the most *urgent* request to the fastest core, which is also the request that has already been executed for the longest time. Intuitively, the longer the system executes the job, the higher probability that the job requires faster cores to complete prior to the deadline. More formally, we define urgency as follows.

Definition 1: Request *urgency* is defined as the expected minimum core speed to complete the request prior to its deadline. Mathematically, we express urgency as follows:

$$\begin{aligned} U &= \frac{\mathbb{E}\{w - w_0 \mid w \geq w_0\}}{r} \\ &= \frac{\int_{w_0}^{\hat{w}} w f(w \mid w \geq w_0) dw - w_0}{r}, \end{aligned} \tag{4}$$

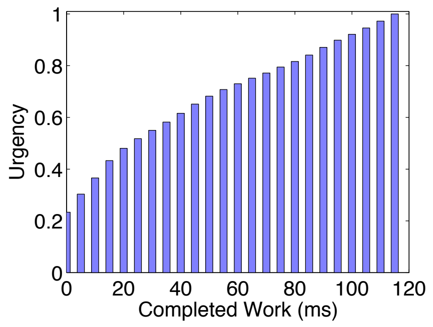


Figure 2: Urgency versus completed work. With more processing, request urgency increases: an older request has higher urgency.

where w_0 is completed work, r is the remaining lifespan of the request and $f(w|w \geq w_0)$ is the PDF of the service demand conditioned on the completed w_0 work.

Urgency indicates the (expected) necessary core speed to complete a job upon its deadline. By assigning faster cores to jobs with higher urgency, jobs have a greater chance to complete prior to their deadlines. Figure 2 shows a lower bound on the urgency value for the Bing service demand distribution (Figure 1(b)), where x-axis is the amount of work that a job has completed and y-axis is urgency. During actual processing, request urgency is impacted by its waiting time in the queue and its execution history, making the urgency in Figure 2 a lower bound.

A key observation is that as the request is processed, its urgency increases. When a request is processed more, its available time before the deadline decreases whereas its expected service demand increases. The general urgency trend is similar for other widely used service demand distributions such as exponential and Pareto. This observation motivates FOF to use faster cores to run the oldest request because that request has high urgency.

3.2 FOF Algorithm

Algorithm 1 shows the pseudo-code of FOF. When a job enters the system, FOF assigns it to the fastest available core. When a job completes or early terminates at its deadline, a core is idle and FOF promotes the oldest job on a slower core to this faster core. No job migrates between cores that have the same speed. Consider the following cases.

All cores are idle FOF assigns the job to the fastest idle core N.

Only the fastest core is busy FOF assigns the new job to idle core $N-1$, the second fastest core in the system.

All cores are busy: When an existing job completes or its deadline expires, a core becomes available. FOF promotes the oldest (longest running) job on any *slower* core to this faster core. It repeats this process until the slowest core becomes idle. At this point, FOF schedules the job at the head of the wait queue on the slowest core.

Algorithm 1 FOF

```
Require: Active job queue $\mathscr{Q}$, core processing speeds $0 < s_1 \leq s_2 \leq \cdots \leq s_N$
 1: Assign urgencies to all jobs: older jobs have higher urgency.
 2: $i \leftarrow N$
 3: while $i \geq 1$ do
 4:     if core $i$ is idle then
 5:         job $\mathscr{J}$ = job being processed on a slower core than core
            $i$ (or waiting in the queue) with the highest urgency
 6:         if job $\mathscr{J}$ is not null then
 7:             schedule job $\mathscr{J}$ to core $i$
 8:         end if
 9:     end if
10:     $i--$
11: end while
12: return
```

The FOF algorithm has the following key properties:

- A faster core always runs a request with higher or equal urgency than all jobs on slower cores, increasing response quality and the probability of completing all requests before their deadlines.
- When there are $1 \leq k < N$ requests where N is the number of cores, the fastest k cores process these k requests, increasing response quality.
- If a request migrates, it always migrates from a slower to a faster core. This choice increases the probability that short jobs will complete on a slow core and that long jobs will execute on fast cores.

Note that FOF is designed to improve quality and throughput on a heterogeneous processor, rather than attain the lowest possible dynamic run-time energy. For example, with only one request in the system, FOF will execute it on the fastest core, whereas executing it on a slower core first will consume less dynamic energy (as shown in Theorem 1). However, by improving throughput while satisfying response quality, the data center can buy and use fewer servers and consume less total energy. Thus, server provisioning and consolidation is the means by which FOF optimizes server and energy cost.

FOF is computationally efficient. It does not require a priori information on each request's *actual* service demand. It also bounds job migrations to $K-1$ times in the worst case, where K is the number of different core speeds, regardless of the server load. In practice, migrations per job are much less than $K-1$, as many short requests are processed and completed on one core.

4 Simulation Study

This section evaluates FOF with a simulation study using Bing web search measurements and various workload distributions. We show heterogeneous processors with FOF improve over homogeneous processors in terms of average quality, quality variance, 95%-quality, and number of servers required to support the workload. Furthermore,

Name	Processor	(C Cores T SMT)	technology	LLC	Speed	1 Core Power	Normalized Performance	Normalized Power
A	i7-2600 Sandy Bridge	(4C 2T)	32 nm	4 M	3.4 GHz	21 W	1.0	1.0
B	i5-670 Nehalem	(2C 2T)	32 nm	8 M	3.4 GHz	16 W	0.92	0.81
C	i7-920 Nehalem	(4C 2T)	45 nm	8 M	2.7 GHz	15 W	0.72	0.73
D	AtomD Bonnell	(2C 2T)	45 nm	1 M	1.7 GHz	4 W	0.45	0.19

Table 1: Core specification, measured and normalized PARSEC performance and power.

FOF achieves a higher quality than various alternative scheduling algorithms, including a clairvoyant scheduler. We explore how to select a good heterogeneous processor configuration based on workload characteristics. We observe that the quality improvement obtained from using FOF on heterogeneous processors translates directly into a throughput increase, thereby reducing the required number of servers. We also explore sensitivity to hardware choice and workload, showing FOF improves throughput at high quality in many scenarios. Section 5 shows how to configure processors with Simultaneous Multithreading (SMT) to mimic heterogeneous processors with FOF and attain benefits in today's data center servers.

4.1 Methodology

As heterogeneous servers are not yet available, we perform a simulation study using DESMO-J, a Java-based discrete-event simulator. The simulator models cores, scheduling, jobs, migration, completion, and other events. Although simulation cannot capture every detail of system implementation, it demonstrates the first-order impact of using FOF on heterogeneous processors. We however validate these simulation results using a finance server executing on an SMT multicore processor.

Workload We use the Bing index search measurements from Section 2.1. We model a server that accepts users' search queries and returns the top L webpages as a CPU intensive process [38]. Our simulator considers a request delay deadline of 120 ms. We model service demand distribution using measurements from production servers and shown in Figure 1(b). We model request arrivals as a Poisson process. To change system load, we control the mean query arrival rate as queries per second (QPS). We use the measured quality profile of Bing web search as shown in Figure 1(a). All these characteristics match other search engines in the literature [48, 35, 38].

Hardware performance and power We approximate heterogeneous core performance and power based on measurements of existing Intel processors. We use the performance and power data reported by Esmaeilzadeh et al. [19] executing PARSEC [7] on four architectures. We use PARSEC because Reddi et al. show that they exhibit similar performance characteristics to interactive services [38]. Table 1 presents core speed, and normalized single core performance and power for four architectures.

Name	A	B	C	D	E	Power
Hom-4A	4	0	0	0	0	82 W
Hom-5B	0	5	0	0	0	81 W
Hom-6C	0	0	6	0	0	88 W
Hom-22D	0	0	0	22	0	82 W
Het-3B-9D	0	3	0	9	0	82 W

Table 2: Processor configurations with design-time power budget.

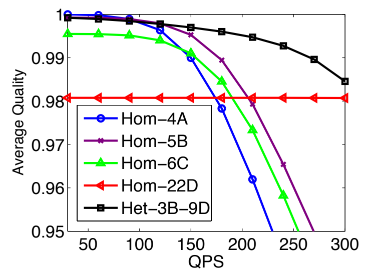
We model homogeneous and heterogeneous processors with a design-time power budget between 80 and 88 W shown in Table 2. Homogeneous configurations include as many individual cores as possible. We simulate a heterogeneous processor composed of three i5 Nehalem cores (3B) and nine AtomD cores (9D), called Het-3B-9D. Section 4.5 explores other heterogeneous configurations.

4.2 Heterogeneous versus Homogeneous

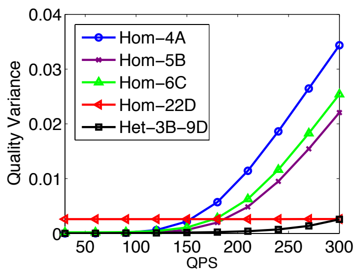
This section shows the benefit of FOF by comparing our default heterogeneous processor (Het-3B-9D) using FOF with the four homogeneous processors from Table 2 using FIFO scheduling in terms of average quality, quality variance, 95%-quality, and number of servers to support a given workload subject to the quality requirement. The widely-used FIFO scheduler places all jobs in a single queue and an idle core pulls the job from the head of the queue and processes it until completion or expiration of the deadline. FOF and FIFO are equivalent on a homogeneous processor.

Improved average quality Figure 3(a) plots average response quality (y-axis) against load measured in QPS (x-axis). A heterogeneous processor with FOF outperforms all homogeneous configurations in terms of average response quality on a wide range of loads, translating into a higher throughput subject to a fixed quality requirement. We focus on throughput at the target quality of 0.99.

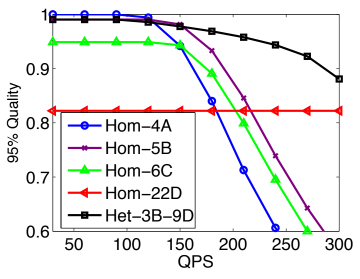
FOF increases throughput significantly, by 50%, on Het-3B-9D compared to Hom-5B, the best 0.99-throughput homogeneous processor. The core utilization at quality 0.99 is as follows: Hom-4A at 150 queries per second (QPS) is 91%; Hom-5B at 180 QPS is 92%; Hom-6C at 155 QPS is 81%; and Het-3B-9D at 270 QPS is 99%. Hom-22D only supports an average quality of approximately 0.9808. At



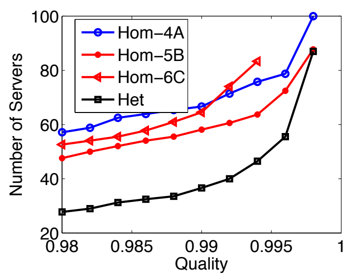
(a) Average quality.



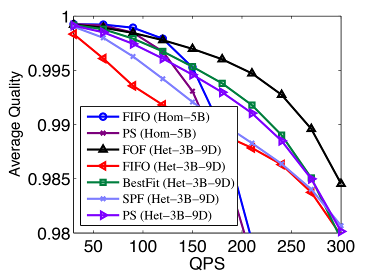
(b) Variance.



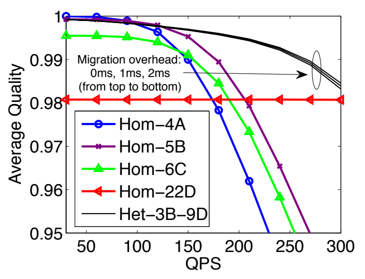
(c) 95%-Quality.



(d) Number of servers.

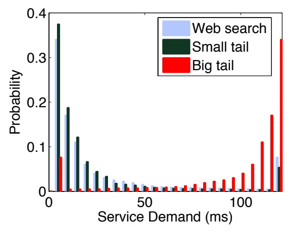


(e) Different scheduling algorithms.

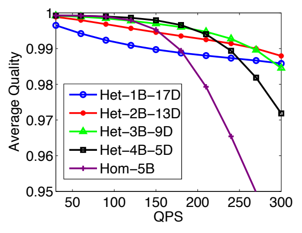


(f) Different migration overheads.

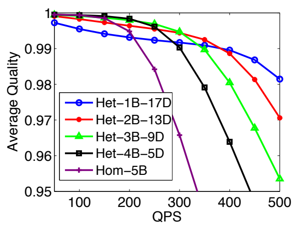
Figure 3: Figures (a), (b), (c) and (d) compare heterogeneous to homogeneous processors under different performance metrics. Figure (e) compares different heterogeneous scheduling algorithms. Figure (f) shows the impact of migration overheads.



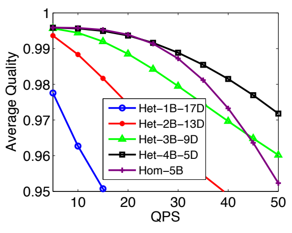
(a) Service demand.



(b) Measured distribution.



(c) Small tail.



(d) Big tail.

Figure 4: Heterogeneous core configurations for a range of service demand distributions.

higher load (not shown), quality drops off further. Figure 3(a) shows that both Hom-4A and Hom-5B, which are fast cores, produce high quality when the throughput is low (e.g., 90 QPS), whereas neither Hom-6C nor Hom-22D are fast enough to achieve sufficiently high quality.

With a fixed power budget, a heterogeneous design satisfies stringent quality requirements (e.g., 0.99) by combining the high processing capabilities of fast cores and the high throughput of multiple low-power slower cores. Key to this result is that one fast core consumes more power than 3 or more slow cores but a fast core has only about 2 times of the processing speed of a slow core.

Reduced quality variance Figure 3(b) shows response quality variance. The heterogeneous processor using FOF has the lowest variance. With Hom-22D, there are enough cores to serve (almost) every incoming job without delay and hence, the quality variance remains relatively constant throughout until the QPS exceeds its capabilities. Nonetheless, long jobs cannot complete prior to their respective deadlines, resulting in a quality variance among short and long jobs. Hom-4A, Hom-5B and Hom-6C have little quality variance with light load (QPS < 150), since they complete almost every job. When load increases, queuing time increases and some long requests get insufficient service before their deadline, which reduces quality and increases variance. When using a Het-3B-9D, a long request that cannot get a fast core immediately can still be processed on one of many slow and medium cores and migrate later to the fast core, which improves quality and reduces variance even at high load.

Improved 95%-quality High-percentile quality is of considerable interest since many commercial services specify their service level agreement (SLA) using both high-percentile quality and average response quality [17]. Remember we compute quality off-line, comparing to a search with no limit on time or resources. For example, a web search engine may target a quality of 0.99 for average quality and 0.90 for at least 95% requests. The high-percentile quality depends on the response quality distribution. Figure 3(c) shows that a heterogeneous processor improves the 95%-quality over homogeneous processors on moderate and heavy loads by improving average quality and reducing variance.

Reducing number of servers To highlight the hardware and energy reductions due to heterogeneous processors with FOF scheduling, we consider a total workload of 10,000 QPS and compute how many servers are needed with a given design-time power budget, subject to various average quality requirements. Figure 3(d) shows the number of servers, for four systems, Het-3B-9D, Hom-4A, Hom-5B and Hom-6C. For average quality of 0.99, a heterogeneous processor reduces the number of servers by approximately 35% compared with homogeneous processors with the same power budget, and may significantly reduce costs and energy usage in large data centers.

4.3 Comparing Scheduling Algorithms

This section compares FOF with four alternative scheduling algorithms: FIFO, Processor Sharing (PS), Slow Preempt Fast (SPF), and BestFit. FIFO, PS, SPF, and FOF are non-clairvoyant because when jobs arrive, their service demand is unknown in practice. BestFit is a clairvoyant scheduler; it knows each request's service demand but lacks migration. Figure 3(e), shows that even without knowledge of the request service demand, FOF outperforms BestFit and all the other schedulers because its migration policy takes advantage of core heterogeneity.

FIFO and EDF Since requests have the same maximum delay, FIFO and Earliest Deadline First (EDF), an optimal real-time scheduler, behave the same. FOF achieves a significantly higher quality than FIFO which cannot support a 0.99-throughput higher than 160 QPS. FOF outperforms FIFO because FOF completes many short requests on slower cores and migrates long requests to faster cores. In FIFO, the assignment of cores does not depend on the request service demand, and processing long requests using slower cores inevitably degrades the total response quality.

Processor sharing (PS) Processor sharing is the well-known round-robin scheduler for homogeneous processors. We extend PS to heterogeneous processors. When the number of jobs M is greater than the number of cores N, the jobs equally share all the available cores. When M is smaller than N, the jobs equally share the M fastest cores in a round-robin fashion. We assume that the context switch and migration overhead of PS is 0 and therefore the PS quality result is an upper bound. Figure 3(e) shows that FOF achieves a higher quality than PS because the FOF migration policy gives long requests a higher chance to use fast cores. PS shares fast cores equally among short and long requests and hence, long requests that really need fast cores may not get them.

These classic scheduling algorithms for homogeneous processors are insufficient because they do not consider how to match request service demands to heterogeneous core characteristics. Moreover, since the request service demand is unknown, it is not possible to determine the most appropriate core for a request before its execution. During execution, however, the scheduler progressively has more information about requests (i.e., urgency) such that the scheduler may refine its decision. Thus, using FOF, *job migration among cores refines and improves scheduling decisions when request service demand is unknown*.

Migration policies We consider the SPF scheduler which migrates jobs in reverse order to FOF, from fast cores to slow ones. Each job is processed until completion or expiration. If the number of jobs is smaller than the number of cores, all the jobs are processed by the fastest available cores. We show in Figure 3(e) that FOF achieves a much higher quality than SPF. By migrating jobs from slower to faster cores, FOF is more likely to complete short jobs on slower cores, saving faster cores to process long jobs. In contrast, SPF completes short jobs on faster cores whereas long jobs, which have higher urgency, are processed by slower cores. Thus, it is likely that long jobs do not get fully completed before their respective deadlines. *This comparison shows that the job migration order from slower to faster cores is critical to exploit processor heterogeneity.*

Clairvoyant without migration Even with known service demands, scheduling multiple jobs on a heterogeneous processor is a challenging task. BestFit tries to schedule each job with the minimum energy in a greedy fashion. Specifically, each core maintains a separate queue and, a new job joins the queue served by the slowest core that can complete the job before its deadline. If none of the cores can complete the job because of a large number of waiting jobs, the job will be scheduled to the queue that produces the highest quality for the job. Hence, BestFit is a greedy clairvoyant scheduler with known service demands but without job migration. Combining job migration with clairvoyance requires solving an integer programming problem that we leave for future work. This algorithm is similar to scheduling algorithms in prior work [14, 40, 44], which map jobs to the most "appropriate" cores.

Figure 3(e) shows that, rather surprisingly, FOF without knowing request service demand outperforms BestFit with known service demand. Because BestFit does not consider job migration, it cannot fully exploit heterogeneous cores. For example, if a long job arrives and the best core to run the job is a fast core but all fast cores are running another job. BestFit will let the job wait for the fast core to finish even when there are other cores available. FOF is better because it uses the slow cores to run the job first and then migrates it to the fast core when it becomes available. FOF also outperforms the Shortest Job First (SJF) algorithm (a widely-used algorithm for improving the response time in homogeneous processors) in terms of response quality. We omit these results due to space limitations. *Even when the request service demand is known, migration is critical to exploiting heterogeneous core resources as they become available. In particular, long requests make progress on slow cores first before migrating when fast cores become available.*

4.4 Migration Overhead

This subsection shows FOF is not sensitive to migration overheads, since actual migration overheads are less than 1%. Even modeling higher migration overheads does not diminish FOF's performance benefits.

Job migration requires a context switch, whose time is proportional to the size of a job's working set due to cache warm-up time, which may take tens of microseconds to a millisecond [18]. We model three migration overhead values: 0 ms (prior section), 1 ms, and 2 ms. Figure 3(f) shows migration overheads hardly have any effect on quality, which is mainly due to the following two factors. (1) Migration overhead is typically at the range of tens of microseconds to a couple of milliseconds and thus is much smaller compared to the deadline of a few hundred milliseconds. (2) The number of migrations is small. Given K different core speeds available, a job may migrate up to $K-1$ times in the worst case, and K is a small number in general. Moreover, as many short jobs complete on slow cores, they do not need to migrate at all. Het-3B-9D has two core types and thus, the upper bound on the number of job migrations is 1 while the average number of migrations per job is 0.45 at 150 QPS and increases to 0.56 at 300 QPS.

4.5 Heterogeneous Core Configuration

This subsection explores how to select good heterogeneous processor configurations based on workload characteristics. We find that more long jobs need more fast cores for a given quality target but some small cores are always required.

We consider three representative workloads as illustrated in Figure 4(a): (1) *measured* distribution of web search; (2) *small tail* reduces the probability of long jobs (> 30 ms) and increases the probability of short jobs (≤ 30 ms) in the measured distribution; and (3) *big tail* reverses the measured distribution (most jobs are long). Figure 4 shows their average quality. We explored more configurations, but for brevity, only show combinations of B and D cores. We compare against a homogeneous processor with B cores, since D cores cannot deliver the desired quality target.

Under all three service demand distributions, a heterogeneous processor outperforms a homogeneous one (Hom-5B) on 0.99-throughput. In particular, Het-1B-17D is the best heterogeneous processor with a small tail, whereas Het-4B-5D is the best under a big tail. Figure 4 confirms our intuition that the best core configurations depend on the service demand distribution and that the more long running jobs the system expects, the more fast cores it should include, because the slow cores cannot produce sufficiently high quality. However, there is always a point where one or more fast cores should be traded for slow cores to gracefully degrade quality and improve throughput.

4.6 Other Workloads

We performed sensitivity studies on synthetic workloads (exponential, Pareto, and bipolar service demand distributions), quality profiles (linear profile, discrete staircase profile), and different deadlines. These results consistently show the benefits of using heterogeneous processors with FOF to meet the desired quality with high throughput. We omit the details of the results due to space constraint.

5 Exploiting Heterogeneity in SMT Systems

This section describes: (a) How to configure an Simultaneous Multithreaded (SMT) multicore processor to act as a heterogeneous processor. (b) How to modify FOF for it. (c) An implementation of a finance server for SMT that delivers improvements in throughput and quality. The experimental results on a 2-way SMT 4 core machine show that FOF-SMT improves throughput by up to 16% compared to the default OS scheduler and 27% compared to no SMT. (d) A comparison of implementation and simulation results confirms the accuracy of our simulator. SMT, as a form of heterogeneity, is already present in data centers, and thus FOF can have an immediate impact.

5.1 SMT as a Heterogeneous Multicore

SMT adds heterogeneity to existing hardware and we use it to mimic heterogeneous processors. Intuitively, a core executing one thread acts as a fast core, and a core executing $N > 1$ SMT hardware threads acts as N slow cores. The N slow cores exhibit higher throughput but each thread runs slower than the fast core, similar to heterogeneous processors.

5.2 FOF for SMT

We modify FOF to create FOF-SMT. We enable SMT for all cores on a processor and then FOF-SMT controls which cores execute as fast cores by executing only one thread or as slow cores by executing N threads on an N-way SMT. FOF-SMT works as follows.

Fast first When a request joins, if there is an idle core, FOF-SMT schedules the request on it.

Fast old Consider two cases. (1) When a request joins, if there are available SMT hardware threads but no idle cores, FOF-SMT schedules the current request on an SMT thread such that it shares the same core as the *youngest* executing request. (2) At the point a request completes, its core may become idle. In this case, if other cores are sharing, FOF-SMT finds the sharing core with the oldest job. Rather than moving the oldest of the jobs sharing a core, it moves the other job. This other job is less urgent, which minimizes the impact on the oldest job and with 2-way SMT, gives both jobs a core to themselves, i.e., fast cores. Both cases leave the old request running on a fast core as they are likely more urgent.

The implementation uses thread pools, affinity, and request queues on each core prioritized by age to schedule jobs in constant time. To study FOF in an implementation and validate our simulator, we use an interactive finance server.

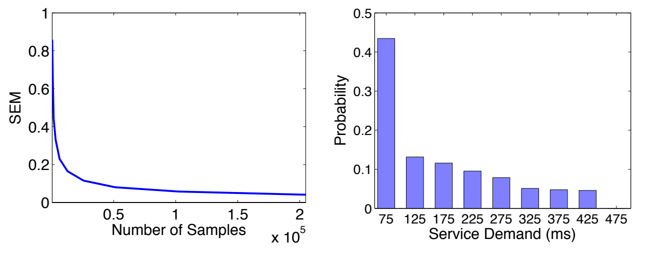
5.3 Option Pricing Finance Server

Finance servers are another example of interactive online services. Banks and fund management companies evaluate thousands of financial derivatives every day, submitting requests that value derivatives and then making immediate trading decisions. Many of these calculations use Monte Carlo methods, which are computationally intensive and rely on repeated random sampling to compute results. We implement an option pricing server that uses Monte Carlo methods for complex path-dependent Asian options [9, 16].

Each request is a Monte Carlo task that estimates an option price under various economic scenarios with different interest rates, strike prices, dividend yields, and volatility. The tasks are time bounded, since traders use them to perform online trading. With more samples, the processing time is longer and the estimated price gets closer to the real price. The system is adaptive and supports returning partial results. The result quality is measured by a well-known statistical metric called standard error of mean (SEM). SEM is the standard deviation of the sample mean of the population mean. SEM is computed by sampling the standard deviation s divided by the square root of the sample size n, i.e., SEM $= s/\sqrt{n}$. The smaller the SEM is, the closer the estimated price to the real price. The goal is to minimize the average and high-percentile SEM value for all requests so the estimated prices are closer to the real prices.

Figure 5(a) shows an error profile with increasing sample sizes. When the number of samples increases along with the processing time, SEM decreases, which indicates increasing quality and the error profile is convex. When the sample size is large, the change in sample standard deviation is small and the square root of the sample size dominates. The convexity of $1/\sqrt{n}$ leads to the convexity of SEM. Minimizing SEM with a convex error profile is equivalent to maximize quality with a concave quality profile.

Figure 5(b) shows the service demand distribution. Each request incurs different processing time to compute a sample, therefore service demand varies. Similar to web search (Figure 1(a)), this workload is non-uniform with a mix of mostly short and some long requests. However, the finance server does not have as heavy of a tail.



(a) Error profile. (b) Service demand distribution.

Figure 5: Measured quality of finance server workloads.

5.4 Methodology

We use a server with 6-core 2-way SMT 3.33 GHz Intel Xeon X5680 processor with 24 GB of memory running Windows Server 2012. The requests follow a Poisson arrival process. When a request's SEM reaches the target of 0.05 or lower, we consider it fully evaluated and terminate the request. Otherwise, the request is partially evaluated and terminated when it reaches a 500 ms deadline. We compare three finance server configurations:

NoSMT SMT disabled.

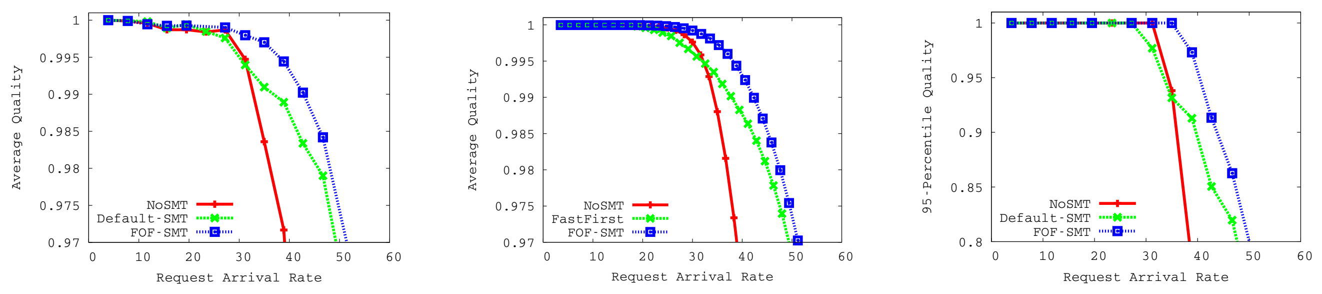
Default-SMT SMT enabled with default round-robin OS scheduling. After all cores are occupied with a single request, it shares with SMT, choosing the core in round-robin fashion [39].

FOF-SMT SMT enabled with FOF-SMT scheduling.

5.5 Implementation Results

We compare FOF with NoSMT and Default-SMT with respect to average and high-percentile quality. Our results show that FOF improves response quality of requests and improves throughput by 27% over NoSMT and 16% over SMT using default round-robin OS scheduling.

Figure 6(a) presents the average quality of requests with varying load. The x-axis is load, expressed as request arrival rate in requests per second (RPS), and the y-axis is the average quality of all requests, where request quality is computed by how far the result is from the target accuracy. The request quality is $1-(SEM_m - SEM_t)/SEM_t$ where SEM_m is the measured SEM value and SEM_t is the target SEM value. The results show that, at light load, all of NoSMT, FOF-SMT and Default-SMT achieve high quality because no core needs to share. With increasing load, FOF-SMT outperforms both NoSMT and Default-SMT with improved quality. For example, at quality target 0.99, NoSMT sustains a throughput of 33 RPS, Default-SMT 37 RPS, and FOF-SMT improves it to 42 RPS, achieving a 27% improvement over NoSMT and 16% over Default-SMT. FOF-SMT reduces quality variance (not shown) and



(a) Implementation: average quality. (b) Simulation: average quality. (c) Implementation: 95-% quality.

Figure 6: Implementation results with 2-way SMT dynamic heterogeneity on 6 cores for a finance server. Implementation matches simulation results. FOF-SMT delivers higher average and 95-percentile quality at higher load.

Figure 6(c) shows that FOF-SMT improves high-percentile quality.

These improvements come from two sources: (1) capacity due to adding hardware parallelism in the form of SMT, and (2) better scheduling choices by FOF. To put capacity improvements in context, we measure a request X alone on a core with processing time T_X and two identical requests on two SMT threads sharing the same core concurrently, which takes time $1.582T_X$. In theory, SMT could improve performance by $2T_X$, but in practice SMT delivers less because it shares hardware resources (issue queues, caches, etc.). In other words, given a core with speed 1, each 2-way SMT hardware context has speed 0.632. 2-way SMT thus provides a $0.632 \times 2 - 1 = 26.4\%$ increase of computational capacity. Thus the capacity improvements of FOF-SMT are close to optimal for this workload. Adding a larger number of slow cores to replace one or more fast cores on heterogeneous processors increases throughput. FOF-SMT makes better scheduling decisions as witnessed by the gap between Default-SMT and FOF-SMT. Smart scheduling algorithms are necessary to fully exploit the benefits of SMT. FOF-SMT outperforms Default-SMT because FOF-SMT shares cores among new requests, which are both likely short and complete on shared (slow) cores, leaving long requests to run on unshared (fast) cores, where they are more likely to complete before the deadline.

5.6 Validating Simulation with Implementation

We validate the simulation results with the finance server implementation. The simulator uses measured service demand, error profile, raw performance on one core (1), and the relative performance of SMT (0.632 of one core) using the 6-core 2-way SMT processor measurements. Comparing Figure 6(b) with the simulation results to the implementation results in Figure 6(a) for average quality shows that the SMT performance reported by the simulator is very close to the SMT implementation results, both with respect to the trends and absolute performance. This result increases our confidence in the accuracy of our simulator and the results in Section 4.

6 Related Work

Heterogeneous processors There are several proposals for heterogeneous processors [27, 4, 31, 43, 5, 40, 14]. ARM recently announced their big.LITTLE processor for production [22], which combines high-performance and energy-efficient cores. Recent work argues for the benefits of heterogeneous processors compared to homogeneous processors in two main scenarios.

(1) A single job has different phases [43, 31, 42], such as parallel phases and sequential phases. This work is grounded in applying Amdahl's law to a parallel program to accelerate its sequential bottleneck [2, 27, 46]. Using a heterogeneous processor, the sequential phase is executed on a high-performance core, and the parallel phase is executed on a number of energy-efficient cores. Our work considers a stream of independent jobs that execute in parallel instead of a single parallel program to which Amdahl's law was applied. We show more than one fast core is often necessary and we furthermore show how to use workload characteristics to choose the mix and variety of fast and slow cores.

(2) A heterogeneous processor is more suitable for multiprogramming environments with diverse application demands [22, 5, 40, 14, 30, 33, 44]. Suleman et al. use high-performance but energy-inefficient fast cores to process critical phases of a job [42]. Others try to match program phases to the appropriate core such that the part of the program that benefits most from the high power core executes on it [5, 40, 44]. They either improve performance or save dynamic energy while being performance neutral. Users run delay-sensitive tasks such as gaming and web surfing using fast cores, while background services such as indexing and spell-checking use slow cores [22]. Lakshminarayana et al. schedule the thread in a parallel job with a larger remaining execution time on a fast core [32]. They predict the remaining time based on thread creation or dynamic profiling. FOF achieves a much more substantial throughput improvement because it leverages the diversity (short versus long jobs) and adaptivity in the workload

demand, since it terminiates a job early if it exceeds it deadline. A long running job cannot monopolize the fast core indefinitely. Moreover, instead of using information about each job, our work exploits the service demand distribution of all jobs.

Real-time scheduling Prior work on real-time scheduling that investigates saving energy while meeting deadlines [47, 1] assumes known service demands, which is not applicable in our environment. Other related work assumes unknown service demand [34, 45, 49] and uses Dynamic voltage/frequency scaling (DVFS) to save energy. None of the prior work considers scheduling multiple requests that both support partial execution as well as share and compete for CPU resources. Trading resource consumption for service quality has also been explored in other contexts (e.g., wireless networks) using stochastic control techniques, but only average queue length or average response time is addressed [37]. Embedded and real-time systems did not explore partial execution, and hence their algorithms are not applicable to interactive services we study. Our objective is to improve total quality with deadline constraints, rather than meeting the deadline of each job.

Next, we discuss scheduling algorithms for SMT and DVFS systems.

SMT In a multiprogrammed environment, prior work on SMT scheduling improves fairness and throughput by coscheduling jobs according to their performance characteristics and interference [41, 20, 12] . They all consider workloads without deadlines. SMT scheduling on real-time and soft real-time systems considers deadlines, but it focuses on periodic tasks such as multimedia applications and partitions resources to meet deadlines [29, 11]. In contrast, we use SMT to emulate a heterogeneous processor and develop an SMT-aware scheduler for interactive workloads, where jobs arrive aperiodically and can be partially evaluated.

DVFS DVFS trades performance for power consumption by adjusting voltage or frequency [47, 1, 21, 34, 45, 49, 13]. Instead of optimizing for dynamic energy, our scheduling improves response quality and therefore supports higher throughput per server under design-time power constraints. Two proposals [34, 45] progressively accelerate the processor speed during job execution to minimize the expected energy based on the service demand distribution, which is consistent with the findings of Theorem 1. However, those studies do not address multiple jobs that compete for resources. Another approach [49] minimizes the energy consumption for multiple types of periodically-arriving jobs, which is not applicable for interactive applications. Similarly, others [15] propose a dynamic voltage scaling algorithm for multimedia applications based on the service demand information provided by content providers, but such information is not available for our applications. Other related work exploits partial execution (referred as differential service level) and proposes an algorithm based on Markov decision process to maximize total response quality given a mean response time constraint [13]. Their algorithm is applicable to server systems with different speeds. They consider mean response time of jobs as constraint but our jobs need to meet response deadlines.

DVFS and heterogeneous processors are complementary technologies. The actual power-performance characteristics of a core depends factors such as pipeline structure, type of transistors, degree of speculation, voltage and frequency. These factors limit the energy efficiency of DVFS at lower speeds and frequencies [6, 31]. In contrast, heterogeneous processors address such inefficiencies by using cores with different microarchitectures to achieve a better tradeoff between performance and energy [22].

While DVFS and SMT are relatively mature technologies that do not require major changes to existing software to exploit their benefits, heterogeneous processors are an emerging technology that will require additional support from the OS, compiler, and libraries before it is practical to use by real-world services and applications.

7 Conclusion

We propose an online scheduling algorithm, FOF, to improve the quality and throughput of an interactive service on a heterogeneous processor. FOF effectively schedules long requests to fast cores and short requests to slow cores without knowing the actual service demands. Extensive simulations evaluate FOF based on workloads from Bing search. The results show that using FOF on heterogeneous processors improves throughput by up to 50% compared to using homogeneous processors. We also show how to use an SMT processor as a dynamic heterogeneous processor. We implement the scheduling algorithm for a financial server. Our experimental results show up to 16% higher throughput by using FOF on an SMT processor compared to a default round-robin OS scheduler. A comparison of the implementation results to our simulator configured with similar features show that they closely match. These results point to significant opportunities for heterogeneous processors in data centers.

8 Acknowledgments

We thank Gregg McKnight and James Larus from Microsoft for the insightful discussions and feedback.

References

[1] S. Albers, F. Muller, and S. Schmelzer. Speed scaling on parallel processors. In *SPAA*, 2007.

[2] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[3] ARM Corporation. big.LITTLE processing, 2011.

[4] S. Balakrishnan, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA*, 2005.

[5] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *ACM Computing Frontiers*, 2006.

[6] M. Bi, I. Crk, and C. Gniady. Iadvs: On-demand performance for interactive applications. In *HPCA*, 2010.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[8] S. Borkar and A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[9] M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Manage. Sci.*, 42:269–285, February 1996.

[10] B. Cahoon and K. S. McKinley. Performance evaluation of a distributed architecture for information retrieval. In *SIGIR*, pages 110–118, Geneva, Switzerland, Aug. 1996.

[11] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Architectural support for real-time task scheduling in smt processors. In *CASES*, 2005.

[12] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. Qos for high-performance smt processors in embedded systems. In *Micro*, 2004.

[13] S. Chaitanya, B. Urgaonkar, and A. Sivasubramaniam. QDSL: QoS-aware systems with differential service levels. *ACM SIGMETRICS*, 2008.

[14] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *DAC*, 2009.

[15] E. Y. Chung, L. Benini, and G. D. Micheli. Contents provider assisted dynamic voltage scaling for low energy multimedia applications. In *IEEE Symposium on Low Power Electronics and Design*, ISPLED'02, 2002.

[16] G. Cortazar, M. Gravet, and J. Urzua. The valuation of multidimensional american real options using the LSM simulation method. *Computers and Operations Research.*, 2006.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[18] C. L. C. Ding and K. Shen. Quantifying the cost of context switch. In *ECS*, 2007.

[19] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ASPLOS*, pages 319–332, 2011.

[20] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *ASPLOS '10*, 2010.

[21] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. Bletsch. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. *Journal of Parallel and Distributed Computing*, 68(9):1175–1185, Sep. 2008.

[22] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM Whitepaper*, 2011.

[23] J. Hamilton. Blog article: Perspectives, 2009.

[24] M. Harchol-Balter. The effect of heavy-tailed job size distributions on computer system design. In *Applications of Heavy Tailed Distributions in Economics*, 1999.

[25] M. Harchol-Balter. Task assignment with unknown duration. *J. of ACM*, 49(2):260–288, 2002.

[26] Y. He, S. Elnikety, and H. Sun. Tians scheduling: Using partial processing in best-effort applications. In *ICDCS*, 20011.

[27] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.

[28] C. Huang, P. A. Chou, and A. Klemets. Optimal control of multiple bit rates for streaming media. In *PCS*, 2004.

[29] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *RTSS*, 2002.

[30] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, 2010.

[31] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, 2003.

[32] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *ACM/IEEE Conference on High Performance Computing (SC)*, 2009.

[33] T. Li, P. Brett, R. C. Knauerhase, D. A. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA*, pages 1–12, 2010.

[34] J. R. Lorch and A. J. Smit. Improving dynamic voltage scaling algorithms with pace. In *SIGMETRICS*, 2001.

[35] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ACM/IEEE International Symposium on Computer Architecture*, ISCA '11, pages 319–330, 2011.

[36] T. Y. Morad, U. C. Weiser, A. Kolodnyt, M. Valero, and E. Ayguadé. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5(1):14–17, 2006.

[37] M. J. Neely. *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Morgan & Claypool, 2010.

[38] V. J. Reddi, B. C. Lee, T. M. Chilimbi, and K. Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *ISCA*, 2010.

[39] M. E. Russinovich, D. A. Solomon, and A. Lonescu. *Microsoft Windows Internals, Fifth Edition: Covering Windows Server 2008, Windows Vista*. Microsoft Press, 2009.

[40] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71(1):114–131, 2011.

[41] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[42] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

[43] M. A. Suleman, Y. N. Patt, E. Sprangle, A. Rohillah, A. Ghuloum, and D. Carmean. Asymmetric chip multiprocessors: Balancing hardware efficiency and programmer efficiency. Technical report, HPS, 2007.

[44] K. Van Craeynest, A. Jalelle, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ISCA '12*, 2012.

[45] R. Xu, C. Xi, R. Melhem, and D. Moss. Practical pace for embedded systems. In *Embedded Software*, 2004.

[46] E. Yao, Y. Bao, G. Tan, and M. Chen. Extending Amdahl's law in the multicore era. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):24–26, Oct. 2009.

[47] F. F. Yao, A. J. Demers, and S. J. Shenker. A scheduling model for reduced CPU energy. In *FOCS*, 1995.

[48] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: a study of yahoo! mobile search queries. In *ACM International Conference on World Wide Web*, WWW '08, pages 257–266, 2008.

[49] W. Yuan and K. Nahrstedt. Energy-efficient CPU scheduling for multimedia applications. *ACM Trans. Computer Systems*, 24(3):292–331, 2006.

Autonomic Management of Dynamically Partially Reconfigurable FPGA Architectures using Discrete Control

Xin An, Eric Rutten
INRIA, Grenoble, France (xin.an@inria.fr, eric.rutten@inria.fr)

Jean-Philippe Diguet, Nicolas le Griguer
Lab-STICC, Lorient, France (jean-philippe.diguet@univ-ubs.fr, nicolas.le-griguer@univ-ubs.fr)

Abdoulaye Gamatié
LIRMM, Montpellier, France (abdoulaye.gamatie@lirmm.fr)

Abstract

This paper targets the autonomic management of dynamically partially reconfigurable hardware architectures based on FPGAs. Discrete Control modelled with Labelled Transition Systems is employed to model the considered behaviours of the computing system and derive a controller for the control objective enforcement. We consider system application described as task graphs and FPGA as a set of reconfigurable areas that can be dynamically partially reconfigured to execute tasks. The computation of an autonomic manager is encoded as a Discrete Controller Synthesis problem w.r.t. multiple constraints and objectives e.g., mutual exclusion of resource uses, power cost minimization.

keywords: Hardware Architectures, Dynamically Partially Reconfigurable FPGA, Discrete Control.

1 Control of autonomic hardware

Controlling FPGAs. We apply the autonomic framework to the context of FPGAs (Field Programmable Gate Arrays), hardware devices that compute a logic function by configuring its gates in a programmable way. A recent progress is *dynamically partially reconfigurable* (DPR) FPGAs. They support partial reconfigurations where only part of gates are reconfigured and reconfigurations to be performed at runtime. Autonomic computing has been seldom applied to such hardware systems, though they represent a significant case of its relevance.

Control for autonomic management. We adopt control techniques to design the *MAPE-K* (Monitor, Analyse, Plan, Execute, based on Knowledge). Formal models are used to describe the possible behaviours of the system under design, and control objectives giving the adaptation policy are specified separately. A controller is then derived based on the system models and objectives. The use of classical control techniques and models, typically these based on continuous time dynamics and differential equations, has been explored for various computing systems [6] and sometimes applied for hardware architectures [5]. A similar approach can be adopted by using *discrete control* techniques, where systems are considered from the viewpoint of events and states. The behaviours can then be modelled in the form of Petri nets or automata for synchronisation [10].

Discrete control for autonomic FPGAs. We apply *discrete control* for the autonomic management of DPR FPGA based embedded systems. A systematic modelling framework is proposed, where system application behaviour, task implementations and executions, architecture reconfigurations and environment are modelled separately by using *Labelled Transition Systems* (LTS) or *automata*. *Discrete Controller Synthesis* (DCS) supported by a programming language and synthesis tool has been applied to compute an autonomic manager.

2 Background notions

2.1 FPGA-based architectures

Basic reconfigurable cell. A FPGA is composed of an array of logic cells and programmable routing channels to implement custom hardware functionalities. A program consists of one or more *bitstream*s, which are binary files storing information to configure logical cells and the routing switches. Recent large FPGAs contain more than 200K logic cells that can be combined and interconnected to implement very complex designs. Multicore architectures with tens of large hardware accelerators and processors can be implemented.

Run-time partial reconfiguration. In the new generation of FPGAs, the hardware configuration can be updated at run-time by using the partial reconfiguration feature. They have the ability to reconfigure hardware during the running of the static part, i.e., the part which does not contain any reconfigurable area. It assumes that the hardware reconfiguration does not disturb the execution

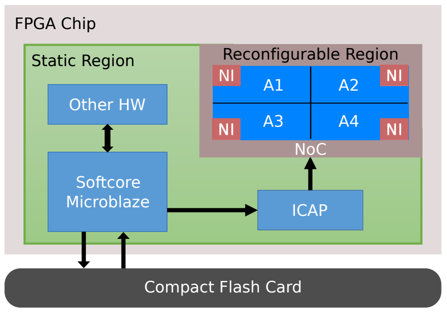


Figure 1: FPGA with a microblaze softcore.

of the application. The bitstreams therefore cover only some regions of the FPGA array.

Such DPR FPGAs make them suitable for addressing constraints on resources (re-using some areas for different functions for applications that can be partitioned into phases) by adapting resources to available parallelism according to environment variations. DPR FPGAs are a trade-off : that they are slower than dedicated Application-Specific Integrated Circuits (ASIC), but much faster than using general purpose CPUs.

Management of reconfiguration. From a technical viewpoint, each hardware configuration file used for the different implementations of the partially reconfigurable regions is stored into a compact flash card. It can be loaded by a processor (e.g. microblaze, which is a 32-bit soft-core processor as implementable on Xilinx FPGAs). It performs the reconfiguration using the ICAP (Internal Configuration Access Port) as in Figure 1.

The runtime management of reconfiguration involves a control loop, taking decision according to events monitored on the architecture, choosing the appropriate next configuration to install, and executing appropriate reconfiguration actions. The architecture dynamism increases the design complexity, for which a complete tool-chain is lacking [8]. Due to the relative novelty of DPR technologies, the management of reconfiguration has to be designed manually for important parts.

Amongst different approaches to address this issue, we investigate the adoption of an autonomic computing approach for the design of reconfiguration control. The MAPE-K structure is based on behavioural models (in the form of automata) for the knowledge about the reconfigurability of these hardware platforms, and discrete control techniques for designing the adaptation policies.

2.2 Discrete control

We consider the modeling framework [1] based on *labelled transition systems* (LTSs) and their parallel composition. LTSs are defined by a finite set of states, between which there are transitions (from source state to target state) with a label of form `c / a`: a firing condition `c` and an action `a`. When a LTS is in some state, if there is a transition for which the condition is true, then it is taken and the next state will be its target state. At the same time the action part will take the value `true`. Two or more LTSs can be composed (noted formally by "|"), representing that they run in parallel: one global step corresponds to one local step for every LTS.

The formalism of LTSs can be used to apply *discrete controller synthesis* (DCS), a formal operation on automata [3, 7]. DCS is an automatic and constructive method to ensure required properties on system behaviors. It applies to an LTS (originally uncontrolled), where inputs $\mathscr{I}$ are partitioned into two subsets, $\mathscr{I}_u$ and $\mathscr{I}_c$, the *uncontrollable* and *controllable* inputs. It takes into account some *control objectives*: properties that must be enforced by control. A controller is synthesized automatically, if it exists, from given LTS's and *objectives*, by applying appropriate algorithms [7] (not detailed here). Its purpose is to constrain the values of controllable variables, in function of states and of uncontrollable inputs, such that system behaviors satisfy the given objectives. The controller is *maximally permissive*, meaning that it allows the largest possible set of correct behaviors.

2.3 Discrete control as MAPE-K

Figure 2(a) shows the MAPE-K architecture of an autonomic system with a loop defining basic notions of Managed Element (ME) and Autonomic Manager (AM). The managed element, system or resource is monitored through sensors. An analysis of this information is used, in combination with knowledge about the system, to plan and decide upon actions. These reconfiguration operations are executed, using as actuators the administration functions offered by the system API. Self-management issues include self-configuration, self-optimisation, self-healing (fault tolerance and repair), and self-protection.

Autonomic managers work in closed loop: for this, one design methodology is to apply techniques from

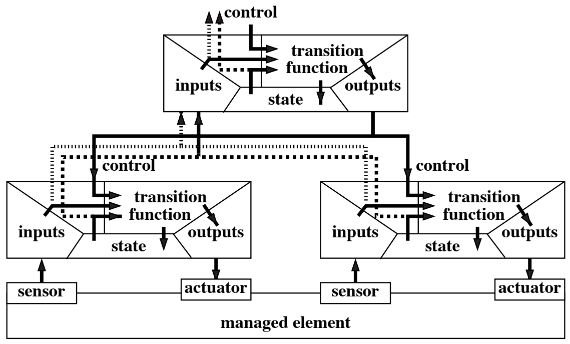


Figure 3: Autonomic coordination for multiple AMs.

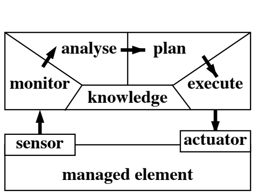
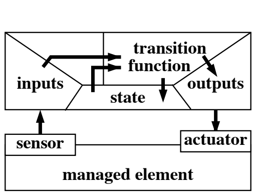
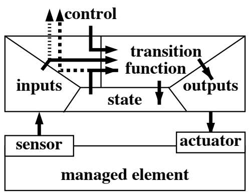
(a)  (b)  (c) 

Figure 2: Autonomic system: (a) the MAPE-K manager; (b) FSM autonomic manager; (c) controllable AM.

Control Theory [6], with the advantage of ensuring interesting properties on the resulting behaviour of the controlled system e.g., stability, convergence, reachability or avoidance of some evolutions. In most cases, continuous models are used, typically for quantitative aspects. More recently, some works relied on Discrete Event Systems (DES), using supervisory control [3], typically for logical or synchronisation purposes e.g., deadlock avoidance in multithreaded programs [10]. They are based on reactive systems models such as Petri nets or Finite State Machines (FSM), which we also call automata. As shown in Figure 2(b), this instantiates the general autonomic loop with knowledge on possible behaviours represented as a formal state machine, and planning and execution in the form of the automaton transition function with a control output, which will trigger the actuator.

Basic features required for a system to be managed in an autonomic fashion have been identified in previous work e.g., in the context of component-based autonomic management [9]: for an ME to be manageable it must be observable and controllable. The manager transforms flows of observations into flows of control choices and actions. Observability translates into outputs, as shown by dashed arrows in Figure 2(c) for an FSM AM, exhibiting (some) of the knowledge and sensor information (raw, or analysed); this can feature state information on the AM itself or of MEs below. Controllability translates to having the AM accept some influence on the decision, and it corresponds to additional input for control, as in Figure 2 for an FSM AM. Its values can be used in the guards and exhibit choices between different transitions.

This builds up to a hierarchical framework as in the structure shown in Figure 3. Given that AMs have been made observable and controllable, an upper-level AM can perform their coordination using their additional control input to enforce a policy. Considering the case of FSM managers makes it possible to encode the coordination problem as a DCS problem. The controller of this upper-level AM is synthesised by DCS.

3 DCS for managing DPR architectures

We present the computing systems of interest through an illustrative example, first informally, then in the model.

3.1 DPR FPGAs

Hardware architecture. We consider a multiprocessor architecture implemented on an FPGA chip (see Figure 1), which includes a general purpose processor: Softcore Microblaze, and a reconfigurable area divided into four tiles: $A1$–$A4$. The communications between architecture components are achieved by a *Network-on-Chip* (NoC). Each processor and reconfigurable tile implements a NoC Interface (NI). Reconfigurable tiles can be combined and configured to implement and execute tasks by loading predefined bitstreams.

The architecture is equipped with a battery supplying the platform with energy. Regarding power management, an unused reconfigurable tile Ai can be put into sleep mode with a *clock gated mechanism* such that it consumes a minimum static power.

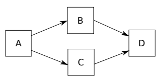
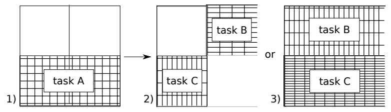
a)  b) 

Figure 4: a) DAG application specification, and b) System configurations and reconfiguration.

Application software. We consider system functionality described as a *directed, acyclic task graph* (DAG). A DAG consists of a set of *nodes* representing the set of tasks to be executed, and a set of directed *edges* representing the precedence constraints between tasks. Figure 4(a) shows an example consisting of four tasks.

In our framework, we suppose each task performs its computation with the following four control points:

- *being requested* or invoked;
- *being delayed*: requested but not yet executed;
- *being executed*: to be executed on the architecture;
- *notifying execution finish*, once it reaches its end.

Occurrences of control points *being requested* and *notifying finishes* depend on runtime situations, and are thus unpredictable and uncontrollable. The way of *delaying* and *executing* tasks is taken charge by a runtime manager aiming to achieve system objectives.

Task implementation. Given a hardware architecture, a task can be implemented in various ways characterised by various parameters of interest, such as used reconfigurable tiles (*ur*), worst case execution time (WCET) (*wt*), and power peak *pp*. For example, two implementations of task *A* can be:

- *A* on *A*1: $wt = 50$, $pp = 20$;
- *A* on $A3 + A4$: $wt = 10$, $pp = 30$;

In this preliminary work, we assume that WCET represents the time cost induced from the start of bitstream loading to the end of task execution. Among possible task implementations, a runtime manager is in charge of choosing the best according to system objectives.

System reconfiguration. Figure 4(b) shows three system configuration examples. In configuration 1, task A is running on tiles A3 and A4 while tiles A and B are set to the sleep mode. Configurations 2 and 3 show two scenarios with tasks B and C running in parallel. Once task A finishes its execution according to the graph of Figure 4(a), the system can go to either configuration 2 or configuration 3 depending on the system requirements. For example, if the current state of the battery level is low, the system would choose configuration 2 as configuration 3 requires the complete FPGA working surface and therefore consumes more power.

System objectives. System objectives define the system functional and non-functional requirements. This section gives the objectives considered in the paper, and categorises them as logical and optimal control objectives. Generally speaking, logical objectives concern state exclusions, whereas optimal objectives target the states associated with optimal costs. Considered logical and optimal control objectives are as follows:

1. resource usage constraint: exclusive uses of reconfigurable areas *A*1-*A*4;
2. energy reduction constraint: switch areas to sleep mode when executing no task;
3. power peak constraint: power peak of hardware platform is constrained w.r.t battery levels;
4. minimise power peak of hardware platform.

More system objectives can be addressed in our framework. We refer the readers to [2] for more details.

3.2 System modelling as a DCS problem

We specify the modelling of the computing system behaviour and control in terms of labelled automata. System objectives are defined based on the models. We focus on the management of computations on the reconfigurable tiles and dedicate the processor area *A*0 exclusively to the resulting controller.

Architecture behaviour. The architecture (see Figure 1) includes a processor, four reconfigurable tiles $\{A1, A2, A3, A4\}$ and a battery. Each tile has two execution modes, and the mode switches are controllable. Figure 5(a) gives the model of the behaviour of tile *Ai*. The mode switch action between Sleep (*Sle*) and Active (*Act*) depends on the value of the Boolean controllable variable c_a_i. The output act_i represents its current mode.

The battery behaviour is captured by the automaton in Figure 5(b). It has three states labelled as follows: *H* (high), *M* (medium) and *L* (low). The model takes input from the battery sensor, which emits level *up* and *down* events, and keeps track of the current battery level through output *st*.

Application behaviour. Software application is described as a DAG, which specifies the tasks to be executed and their execution sequences and parallelism. Its execution behaviour can be captured by using an automaton with states representing the set of tasks that are active in current states. The firing conditions of transitions are task *finish notifications*, which could enable the executions of (some of) its immediate succeeding tasks by emitting *start* requests of these tasks. An algorithm to systematically construct such an scheduling automaton for a DAG can be found in [2].

Task execution behaviour. In consideration of the four control points of task executions (see Section 3.1), the execution behaviour of task *A* associated with two implementations (see Section 3.1) can be modelled as Figure 5(c). It features an initial *idle* state I_A, a *wait* state W_A, and two *executing* states X_A^1, X_A^2 corresponding to two implementations of task *A*. Controllable variables are integrated in the model to encode the controllable points: being delayed and executed. Upon the receipt of *start* request r_A, task *A* goes to either:

- *executing* state $X_A^i, i \in \{1,2\}$ if the value of *controllable* variable c_i leading to X_A^i is *true*, or
- *wait* state W_A if delayed, i.e., the value of Boolean expression $c = \bigvee c_i, i \in \{1,2\}$ is false.

From wait state W_A, upon the receipt of event c_i, it goes to execution state X_A^i. When the execution of task *A* finishes, i.e., the end notification event e_A is received, the automaton goes back to *idle* state I_A. Output *es* represents its execution state.

Local execution costs. The execution costs of different task implementations are different. Three cost parameters are considered (see Section 3.1). We capture them by associating cost values denoted by a tuple (rs, wt, pp) with the states of task models, where: $rs \in 2^{RA}$ (*RA* is the set of architecture resources), $wt \in \mathbb{N}$ (a WCET value) and $pp \in \mathbb{N}$ (a power peak). The costs associated with

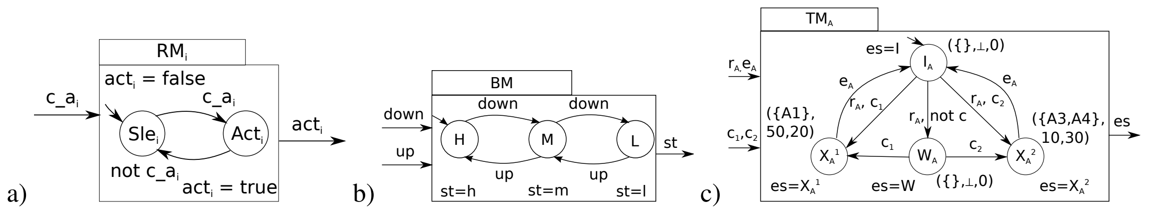


Figure 5: Models RM_i for tile Ai, BM for battery, and TM_A for the execution behavior of task A.

executing states are the values associated with their corresponding implementations. For *idle* and *wait* states, apparently $rs = \emptyset, pp = 0$. However, the wt values for *idle* and *wait* states depend on the execution times of their precedent tasks. We therefore represent it by using a special symbol $\perp$, and thus we have $wt \in \mathbb{N} \cup \perp$.

Global system behaviour model. The parallel composition of control models for reconfigurable tiles RM_1-RM_4, battery BM and tasks TM_A-TM_D, plus scheduler Sdl comprises the system model: $\mathscr{S} = RM_1|...|RM_4|BM|TM_A|...|TM_D|Sdl$ with initial state $q_0 = (Sle_1, ..., Sle_4, H, I_A, ..., I_D, I)$. Sdl represents the automaton that captures the application behavior as discussed in Section 3.2. It represents all the possible system execution behaviours in the absence of control (i.e., a runtime manager is not yet integrated).

Global costs. A system state q is a composition of local states (denoted by $q_1, ..., q_n$), and we define its global cost from the local ones as follows:

- used resources: union of used resources associated with the local states, i.e., $rs(q) = \bigcup rs(q_i), 1 \leq i \leq n$;
- power peak: the sum of values associated with the local states, i.e., $pp(q) = \sum(pp(q_i), 1 \leq i \leq n)$;

System objectives. The two types of system objectives: logical and optimal ones, can then be defined in terms of the states and the costs defined on the states or paths of the model. For example, Objective 1) exclusive uses of reconfigurable areas A1-A4 by tasks is defined by $\forall q_i, q_j \in q, i \neq j$, that $rs(q_i) \bigcap rs(q_j) = \emptyset$. We refer the readers to [2] for the detailed definition.

We have validated our models and manager computations experimentally by implementing a video processing system on an ML605 board from Xilinx containing an FPGA. The BZR language has been used to encode system models and objectives, and generate a correct autonomic manager in C code for the system. They are detailed elsewhere [2] due to lack of space.

4 Conclusion and Perspectives

Reconfigurable architectures, especially DPR FPGAs, constitute a platform for adaptive computing that is gaining widespread use. They are a typical target for autonomic computing approaches, although they are not often explicitly tackled that way. In this paper, we proposed a systematic modeling framework for DPR FPGA based embedded systems, and applied formalisms and tools from discrete control to encode and perform the autonomic manager computation as a DCS problem.

Perspectives include the ongoing work to enrich our models with reconfiguration costs, and the use of modular synthesis and compilation [4] for manager computing. We are working on more experimental systems, which will validate more completely our approach.

References

[1] ALTISEN, K., CLODIC, A., MARANINCHI, F., AND RUTTEN, E. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming (ESOP'03)* (2003), pp. 174–188.

[2] AN, X., RUTTEN, E., DIGUET, J.-P., LE GRIGUER, N., AND GAMATIÉ, A. Autonomic management of reconfigurable embedded systems using discrete control: Application to fpga. Research Report RR-8308, INRIA, May 2013. `http://hal.inria.fr/hal-00824225`.

[3] CASSANDRAS, C., AND LAFORTUNE, S. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 2008.

[4] DELAVAL, G., MARCHAND, H., AND RUTTEN, E. Contracts for modular discrete controller synthesis. In *Conf. on Languages, Compilers, and Tools for Embedded Systems* (2010), pp. 57–66.

[5] EUSTACHE, Y., AND DIGUET, J.-P. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Conf. on Hardware/Software codesign and system synthesis (CODES/ISSS)* (2008), pp. 67–72.

[6] HELLERSTEIN, J., DIAO, Y., PAREKH, S., AND TILBURY, D. *Feedback Control of Computing Systems.* Wiley, 2004.

[7] MARCHAND, H., AND SAMAAN, M. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng. 26*, 8 (2000), 729 –741.

[8] SANTAMBROGIO, M. D. From reconfigurable architectures to self-adaptive autonomic systems. *IJES 4*, 3/4 (2010), 172–181.

[9] SICARD, S., BOYER, F., AND PALMA, N. D. Using components for architecture-based management: the self-repair case. In *Proc. Conf. ICSE* (2008).

[10] WANG, Y., LAFORTUNE, S., KELLY, T., KUDLUR, M., AND MAHLKE, S. The Theory of Deadlock Avoidance via Discrete Control. In *Conf. POPL* (2009).

FMEM: A Fine-grained Memory Estimator for MapReduce Jobs

Lijie Xu[†‡], Jie Liu[†], and Jun Wei[†]

[†]Institute of Software, Chinese Academy of Sciences
[‡]University of Chinese Academy of Sciences
{xulijie09, ljie, wj}@otcaix.iscas.ac.cn

Abstract

MapReduce is designed as a simple and scalable framework for big data processing. Due to the lack of resource usage models, its implementation Hadoop hands over resource planning and optimizing works to users. But users also find difficulty in specifying right resource-related, especially memory-related, configurations without good knowledge of job's memory usage. Modeling memory usage is challenging because there are many influencing factors such as framework's dataflow, user-defined programs, large space of configurations and memory management mechanism of JVM. In order to help both users and the framework to analyze, predict and optimize memory usage, we propose a **F**ine-grained **M**emory **E**stimator for **M**apReduce jobs called FMEM. FMEM contains a dataflow estimator which can predict the data volume flowing among map/reduce tasks. Based on dataflow and rules of memory utilization learnt from a lot of jobs, FMEM uses a rules-statistics method to estimate fine-grained memory usage in each generation of task's JVM. Representative benchmarks show that FMEM can predict diverse jobs' memory usage within 20% relative error. Furthermore, FMEM will be promoted to find optimum dataflow and memory related configurations.

1 Introduction

Google MapReduce [7] framework and its open-source implementation Hadoop have been widely adopted to process big data. This framework divides the costly data processing job into small independent map/reduce tasks and runs them in parallel. Users only need to specify map and reduce functions to develop data-intensive applications, regardless of distributed issues. Users can also write SQL-like scripts which can be transformed into MapReduce jobs automatically by high-level frameworks such as Pig [15], Hive [18] and Sawzall [16].

Although MapReduce helps users focus on job's function implementation, we find its *three-isolated-layer* architecture causes users' difficulty in configuration, resource planning and performance optimization. In *user* layer, users are required to write programs, prepare dataset and also specify appropriate memory-related configurations. In *framework* layer, besides defining job's data processing steps (dataflow [5]) in map and reduce stage, framework is also responsible to schedule, launch and maintain map/reduce tasks. In *execution* layer, each task runs as a separate JVM instance, performs data processing steps and executes concrete map/reduce functions. Since JVM divides memory into small spaces and manages them separately, only execution layer knows the actual fine-grained memory usage. Framework just treats memory as a large contiguous space without modeling its consumption. So inappropriate configurations may lead to job's OutOfMemory error, performance degradation or resource waste. At the highest layer and facing large space of configurations, users usually feel hard to analyze, predict and optimize memory usage. However, new scheduling frameworks such as YARN [6] and Mesos [12] not only require users to specify the memory usage but also schedule tasks according to it.

It is challenging to model and predict job's memory usage with variable dataset, limited logs and large space of configurations. Fortunately, MapReduce dataflow pattern is relatively fixed with only black-box map/reduce functions. Our proposed memory estimator (FMEM) uses simulation method to model dataflow pattern and statistical methods to model intermediate data volume. Memory usage is more complex to model because of multiple factors such as dataflow, configurations and garbage collection (GC). In order to build this model, we integrate the different views of memory consumption in all layers, study the memory management mechanism of JVM, analyze a lot of jobs' logs, and then summarize rules of fine-grained memory usage. Statistical methods are used to estimate the size of in-memory objects. Finally, FMEM profiles a job using sample data and then predict its dataflow and memory usage on *real* big data.

Our contributions are as follows: 1) We provide a detailed analysis of job's memory usage, considering dataflow and memory management from user-level to inner JVM. 2) We also introduce a fine-grained memory estimator which can predict job's memory usage in a large space of configurations.

2 Memory Usage Analysis

Each mapper/reducer runs as an independent process in MapReduce framework. In Hadoop, one process is one JVM instance which isolates the framework from managing the physical memory directly. Memory allocation and GC are controlled by specific algorithms. JVM divides the whole heap space into two parts: *new* generation for storing newly-generated objects and *old* generation for storing long-term objects. We find task's memory consumption mainly comes from the following items:

Memory Buffers. In mapper, spill buffer always occupies a large fixed space in old generation. It is set by *io.sort.mb* and used to cache map() outputs. Enlarging this buffer may reduce spill times and disk I/O. In reducer, data shuffled from map outputs are kept as in-memory segments in a logical shuffle buffer. This buffer cannot exceed a threshold (default 70%) of JVM's total heap, or else segments are merged onto disk. In JVM, segments are first allocated in new generation and some of them are transferred into old generation if GC occurs. In addition, Java's input/output/flush/compress streaming classes contain small-sized buffers.

Records. Since each task has to read <K, V> records, process them, merge intermediate records and output new records, records definitely occupy a large space in JVM. In mapper, map() outputs records into spill buffer. In reducer, shuffled records are first kept as segments in shuffle buffer, though they may be merged onto disk later. Streaming records in map() and reduce() occupy limited space unless many of them are kept purposely into in-memory data structure.

Temporary Objects (TmpObjs). While processing and producing records, user-defined programs or framework itself may generate temporarily referenced objects such as char[], byte[], String, ArrayList and so on. Most of them are auxiliary objects of input/output records, allocated in new generation first and then reclaimed by GC. For example, A WordCount mapper produces massive *java.nio.HeapCharBuffer* objects. Objects' number equals the number of map() output records, but their size is more than 7 times the size of map() input records.

Others. The native libraries used in task's JVM may consume small memory space. JVM also keeps a small area to store programs' Class, Object and Method information. Other program-related items such as code segment and thread pool also have small space in memory.

3 System Overview

To predict jobs' $<$Memory Usage *mu*$>$ under specific $<$Dataset *d*, Configuration *c*$>$, we build an integrated system illustrated in Figure 1. We first profile the sample job running on sample dataset (*SData*) and then estimate big job's *mu* on big dataset (*BData*). *Conf* stands for Configuration.

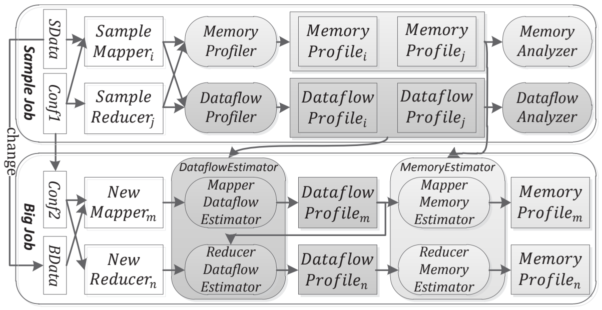


Figure 1: System Architecture of FMEM

Built-in Monitor: To monitor dataflow, we add many fine-grained dataflow counters into Hadoop's task logs. For example, we add each spill piece's Records/Bytes Statistics (RBS) before and after spilling, each partition's RBS before and after merging and so on. We also use *Jstat* [4] to record each generation's memory usage every *N* seconds. Users can turn on or off built-in monitor through configuration. This monitor has low overhead and only used for sample jobs.

Profiler: After a sample job finishes, log collector will fetch each task's execution time, configuration, dataflow volume and memory usage. Dataflow profiler calculates task's RBS in map, spill&merge, shuffle, sort and reduce phase. Similarly, memory profiler calculates max/min/average memory usage in each phase.

Dataflow Estimator: Though we can get RBS from the sample job, it is non-trivial to predict big job's dataflow in a large configuration space. Many configurations such as input split size, spill buffer and reducer number can affect dataflow volume. To tackle them, we actually build a simulator of MapReduce framework to model dataflow in each processing step. Statistical methods are used to model and estimate the I/O ratio. When big job's *BData* and *Conf2* are specified, mapper dataflow model in our simulator uses sample mappers' profiles to estimate new mappers' dataflow. Then, reducer dataflow model can compute new reducers' profiles based on the sample ones.

Memory Estimator: To estimate new tasks' memory profiles, we first compute the size of their memory-consuming items. We get memory buffer size from *Conf2*, get records' size from dataflow estimator, and compute TmpObjs according to dataflow and memory profiles of sample tasks. Next, we use rules summarized from tremendous jobs' profiles to estimate memory usage in each generation for each task. The rules are formalized as $NGU/OGU \approx f\,(Conf, Records, TmpObjs)$. Finally, memory estimator selects the maximum (x) memory usage of all the new mappers to represent mapper's *mu*. In detail, mapper's xNGU represents maximum memory usage in new generation of all mappers. So does reducer's. xOU stands for that in old generation. xHeapU denotes heap usage (i.e., NGU + OU).

4 Evaluation

Because each task runs as an independent JVM instance and processes its own data, task's memory usage is not so sensitive to cluster scale as job's execution time. We evaluate FMEM's accuracy on a local cluster of 10 nodes. Each node has four Intel i7-2600 cores, 16GB RAM and 2TB disk space. OS is Ubuntu-11.04 x86_64 and JDK is HotSpot 64-Bit Server VM (build 1.6.0_27). Hadoop version is 0.20.2 which is similar to the latest 1.1.2. YARN does not change dataflow pattern either. One node act as JobTracker. The others are slave nodes, each of which has 4 map slots and 2 reduce slots.

We use diverse applications (Table 1) to evaluate FMEM. *Combine* denotes whether combine() is used. *Compress* denotes whether spill pieces and segments are compressed. *SeqBlock* means Block compression in SequenceFile. For each application, we run 180 sample jobs (processing 1GB sample dataset) and 180 big jobs (processing big dataset) with different combinations of <split, ismb, RN, Xmx, Xms> (SIRXX). These five configurations are often adjusted to better performance, though our models involve many other configurations. *Split* is input split size (set to 64, 128 or 256MB). *ismb* is *io.sort.mb* (set to 200, 400, 600 or 800MB). Sample jobs' *RN* (reducer number) is 2 or 4, while big jobs' *RN* is 9 or 18. JVM's maximum heap size *Xmx* is set to 1000, 2000, 3000 or 4000MB. Minimum heap size *Xms* is not set or set equal to *Xmx*. So the number of sample/big jobs is 192. Twelve of them are abortive jobs because of memory overflow. Next, we use a sample job with specific <split, ismb, RN, Xmx, Xms> to estimate a big job's *mu* with another SIRXX. So there are 180 * 180 = 32,400 estimated memory usage <*emu*>. Finally, we compare each big job's estimated <*emu*> and real <*rmu*> using relative error as follows:

$$\textit{relative error} = \left| \frac{emu - rmu}{rmu} \right| * 100\%$$

If *rmu* = 0, we set relative error to 100%. The sample job randomly selects several splits (totally 1GB) from all the input splits of big dataset as sample dataset.

Table 1: Representative MapReduce Applications

Applications	Dataset	Combine	Compress
WikiWordCount	9.4 GB	Y	N
BuildInvertedIndex	9.4 GB	N	SeqBlock
UserVisits_Aggre-pig	75 GB	Y	N
TwitterBiEdgeCount	24.4 GB	N	N
TeraSort	36 GB	N	Y

WikiWordCount (WWC): This application uses standard WordCount program from Hadoop Examples. We preprocess enwiki-20110405-pages-articles.xml and get 9.4GB plain text as input big dataset.

BuildInvertedIndex (BII): This application simulates building inverted index of Web pages, which is widely used in search engines. The source code is from [1]. Input dataset is as same as that in WWC.

UserVisits_Aggre-pig (UVA): This application is actually a Pig script which is used to analyze user-visited logs in websites. We get this script from Hive Performance Benchmark in [3]. It has *Group By* operator and uses program-generated dataset.

TwitterBiEdgeCount (TBEC): It counts the number of bilateral edges of Twitter graph from [13]. This large sparse graph has more than 40 million nodes and 1.5 billion edges.

TeraSort (TS): This application also uses standard TeraSort program and sorts program-generated 36 GB dataset. Note that this job uses identity map() and reduce(). Thus, the I/O ratio of them is 1:1.

4.1 Evaluating Memory Estimator

Each job's memory usage is represented by mapper's *mu* and reducer's *mu*. We evaluate them separately. Each histogram in Figure 2 shows the average relative error from 32,400 comparisons of big jobs' <*emu*, *rmu*>. Four metrics (xOU, xNGU, xHeapU and RSS) are used as concrete *mu* for both mappers and reducers. Suppose a big job has *n* mappers, this job's mapper $xOU = max_{1 \leq i \leq n}(OU_i)$. Others are computed in the same way. HeapU represents total memory usage of JVM, while RSS (Resident Set Size) stands for non-swapped physical memory usage in Linux. Sometimes there is a small difference between them. The top part shows mapper's relative error. Compared with xOU, xNGU has higher error rate. One reason is that NGU is more variable and affected by multiple factors. Another is that our estimating condition is very harsh. We only use a single sample job with one configuration to estimate a big job with another configuration. xHeapU and RSS are better but their standard deviations are a little high. The bottom part shows reducer's relative error. Since reducer's *mu* is related to the size of shuffled records, large difference of dataflow may cause high error rate of *mu*. So WWC's xOU and xNGU have high error rate. But for the other applications, xNGU and xOU have low error rates which indicate our memory usage rules are effective.

5 Related Work

Many researchers have studied job's performance model and optimizing methods. Some are concerned about

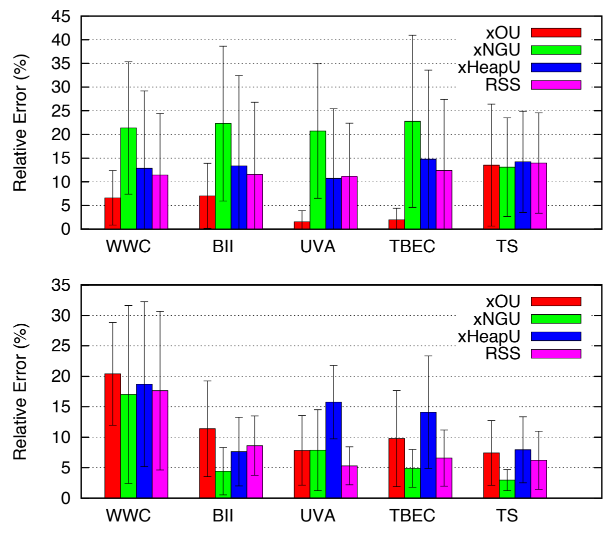


Figure 2: Relative error with standard deviation of <*emu*,*rmu*>

jobs' execution time estimation. Morton et al. [14] present an online heuristic method to predict the progress of MapReduce job pipelines. This method borrows some ideas from progress indicators for SQL queries in DBMSs. Verma et al. [19] propose a theoretical bound time model which analyzes each phase of MapReduce dataflow carefully. They also discuss how to allocate right resource (slots) to guarantee job's runtime [20]. Ganapathi et al. [8] use Kernel Canonical Correlation Analysis to model the relationship between Hive queries' features and queries' performance metrics (only runtime is validated). This method does not focus on the actual MapReduce job and treats the dataflow as a black box.

Other researchers optimize job's configurations. S-tarfish project [11, 10] proposes a cost-based optimizer to find job's optimum configuration. The *What-if* engine in this project can predict job's performance (mainly for runtime) with different configurations. Hadoop performance models are discussed in [9] but fine-grained memory usage is not studied.

Few works focus on job's memory usage. Singer et al. [17] design a fork-join MapReduce Java Framework (MRJ) for multi-core machines. They use machine learning approach to finding most suitable GC policy for MRJ, but memory usage is not studied. This method does not concentrate on distributed MapReduce framework like Hadoop either.

6 Conclusion

Memory is more *precious* compared with disk for big data processing. YARN and Mesos schedule tasks according to CPU and memory requirement. To help users analyze, predict and optimize resource usage, we develop FMEM which can estimate MapReduce job's dataflow and memory usage in a large configuration space. It uses sample job's profiles to estimate big job's resource usage. FMEM models the complex relationship among dataflow, memory usage, GC and configurations. It can also be promoted to tackle other resource-related problems. To the best of our knowledge, this is the first approach that tries to model the memory usage of distributed MapReduce tasks. Our project is now available at github [2].

7 Acknowledgement

We are indebted to our reviewers and Xiulei Qin for their insightful comments. This work is supported by the National Natural Science Foundation of China (61202065, 61173005), the National Grand Fundamental Research 973 Program of China (2009CB320704), and the National High-Tech R&D Plan of China (2012AA011204).

References

[1] Cloud^9. http://lintool.github.com/Cloud9/.
[2] FMEM. https://github.com/JerryLead/FMEM.git.
[3] Hadepot. http://nuage.cs.washington.edu/repository.php.
[4] Jstat. http://tinyurl.com/c5g2kuz.
[5] MapReduce Dataflow. http://tinyurl.com/yzhlkep.
[6] YARN. http://tinyurl.com/bnadg9l.
[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[8] A. Ganapathi et al. Statistics-driven workload modeling for the cloud. In *ICDE Workshops*, 2010.
[9] H. Herodotou. Hadoop performance models. *CoRR*, abs/1106.0940, 2011.
[10] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB*, 4(11):1111–1122, 2011.
[11] H. Herodotou et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
[12] B. Hindman et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
[13] H. Kwak et al. What is twitter, a social network or a news media? In *WWW*, 2010.
[14] K. Morton et al. Estimating the progress of MapReduce pipelines. In *ICDE*, 2010.
[15] C. Olston et al. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.
[17] J. Singer et al. Garbage collection auto-tuning for java MapReduce on multi-cores. In *ISMM*, 2011.
[18] A. Thusoo et al. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.
[19] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic resource inference and allocation for MapReduce environments. In *ICAC*, 2011.
[20] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *Middleware*, 2011.

AGILE: elastic distributed resource scaling for Infrastructure-as-a-Service

Hiep Nguyen, Zhiming Shen, Xiaohui Gu
North Carolina State University
{hcnguye3,zshen5}@ncsu.edu, gu@csc.ncsu.edu

Sethuraman Subbiah
NetApp Inc.
sethu.subbiah@netapp.com

John Wilkes
Google Inc.
johnwilkes@google.com

Abstract

Dynamically adjusting the number of virtual machines (VMs) assigned to a cloud application to keep up with load changes and interference from other uses typically requires detailed application knowledge and an ability to know the future, neither of which are readily available to infrastructure service providers or application owners. The result is that systems need to be over-provisioned (costly), or risk missing their performance Service Level Objectives (SLOs) and have to pay penalties (also costly). AGILE deals with both issues: it uses wavelets to provide a medium-term resource demand prediction with enough lead time to start up new application server instances before performance falls short, and it uses dynamic VM cloning to reduce application startup times. Tests using RUBiS and Google cluster traces show that AGILE can predict varying resource demands over the medium-term with up to 3.42× better true positive rate and 0.34× the false positive rate than existing schemes. Given a target SLO violation rate, AGILE can efficiently handle dynamic application workloads, reducing both penalties and user dissatisfaction.

1 Introduction

Elastic resource provisioning is one of the most attractive features provided by Infrastructure as a Service (IaaS) clouds [2]. Unfortunately, deciding when to get more resources, and how many to get, is hard in the face of dynamically-changing application workloads and service level objectives (SLOs) that need to be met. Existing commercial IaaS clouds such as Amazon EC2 [2] depend on the user to specify the conditions for adding or removing servers. However, workload changes and interference from other co-located applications make this difficult.

Previous work [19, 39] has proposed prediction-driven resource scaling schemes for adjusting how many resources to give to an application within a single host. But distributed resource scaling (e.g., adding or removing servers) is more difficult because of the latencies involved. For example, the mean instantiation latency in Amazon EC2 is around 2 minutes [8], and it may then take a while for the new server instance to warm up: in our experiments, it takes another 2 minutes for a Cassandra server [4] to reach its maximum throughput. Thus, it is insufficient to apply previous short-term (i.e., less than a minute) prediction techniques to the distributed resource scaling system.

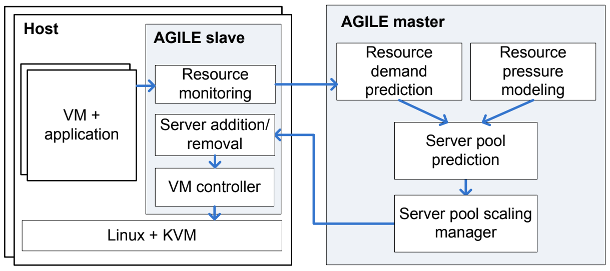


Figure 1: The overall structure of the AGILE system. The AGILE slave continuously monitors the resource usage of different servers running inside local VMs. The AGILE master collects the monitor data to predict future resource demands. The AGILE master maintains a dynamic resource pressure model for each application using online profiling. We use the term *server pool* to refer to the set of application VMs that provide the same replicated service. Based on the resource demand prediction result and the resource pressure model, the AGILE master invokes the server pool manager to add or remove servers.

In this paper, we present our solution: AGILE, a practical elastic distributed resource scaling system for IaaS cloud infrastructures. Figure 1 shows its overall structure. AGILE provides medium-term resource demand predictions for achieving enough time to scale up the server pool before the application SLO is affected by the increasing workload. AGILE leverages pre-copy live cloning to replicate running VMs to achieve immediate performance scale up. In contrast to previous resource demand prediction schemes [19, 18], AGILE can achieve sufficient lead time without sacrificing prediction accuracy or requiring a periodic application workload.

AGILE uses online profiling and polynomial curve fitting to provide a black-box performance model of the application's SLO violation rate for a given resource pressure (i.e., ratio of the total resource demand to the total resource allocation for the server pool). This model is updated dynamically to adapt to environment changes such as workload mix variations, physical hardware changes, or interference from other users. This allows AGILE to derive the proper resource pressure to maintain to meet the application's SLO target.

By combining the medium-term resource demand prediction with the black-box performance model, AGILE can predict whether an application will enter the overload state and how many new servers should be added to avoid this.

Contributions

We make the following contributions in this paper.

- We present a wavelet-based resource demand prediction algorithm that achieves higher prediction accuracy than previous schemes when looking ahead for up to 2 minutes: the time it takes for AGILE to clone a VM.
- We describe a resource pressure model that can determine the amount of resources required to keep an application's SLO violation rate below a target (e.g., 5%).
- We show how these predictions can be used to clone VMs proactively before overloads occur, and how dynamic memory-copy rates can minimize the cost of cloning while still completing the copy in time.

We have implemented AGILE on top of the KVM virtualization platform [27]. We conducted extensive experiments using the RUBiS multi-tier online auction benchmark, the Cassandra key-value store system, and resource usage traces collected on a Google cluster [20]. Our results show that AGILE's wavelet-based resource demand predictor can achieve up to 3.42× better true positive rate and 0.34× the false positive rate than previous schemes on predicting overload states for real workload patterns. AGILE can efficiently handle changing application workloads while meeting target SLO violation rates. The dynamic copy-rate scheme completes the cloning before the application enters the overload state with minimum disturbance to the running system. AGILE is light-weight: its slave modules impose less than 1% CPU overhead.

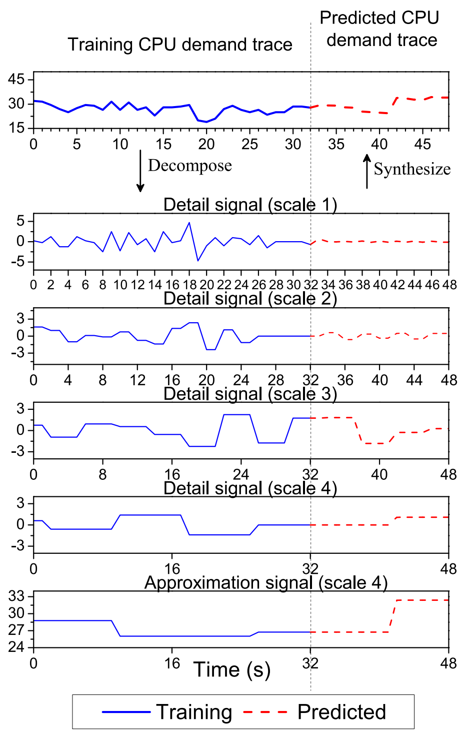


Figure 2: Wavelet decomposition of an Apache web server CPU demand under a real web server workload from the ClarkNet web server [24]. The original signal is decomposed into four detailed signals from scale 1 to 4 and one approximation signal using Haar wavelets. At each scale, the dotted line shows the predicted signal for the next future 16 seconds at time t = 32 second.

2 AGILE system design

In this section, we first describe our medium-term resource demand prediction scheme. By "medium-term", we mean up to 2 minutes (i.e., 60 sampling intervals given a 2-second sampling interval). We then introduce our online resource pressure modeling system for mapping SLO requirements to proper resource allocation. Next, we describe the dynamic server pool scaling mechanism using live VM cloning.

2.1 Medium-Term Resource demand prediction using Wavelets

AGILE provides *online* resource demand prediction using a sliding window D (e.g., $D = 6000$ seconds) of recent resource usage data. AGILE does not require advance application profiling or white-box/grey-box application modeling. Instead, it employs *wavelet transforms* [1] to make its medium-term predictions: at each sampling instant t, predicting the resource demand over the prediction window of length W (e.g., $W = 120$

seconds). The basic idea is to first decompose the original resource demand time series into a set of wavelet based signals. We then perform predictions for each decomposed signal separately. Finally, we synthesize the future resource demand by adding up all the individual signal predictions. Figure 2 illustrates our wavelet-based prediction results for an Apache web server's CPU demand trace.

Wavelet transforms decompose a signal into a set of wavelets at increasing scales. Wavelets at higher scales have larger duration, representing the original signal at coarser granularities. Each scale i corresponds to a wavelet duration of L_i seconds, typically $L_i = 2^i$. For example, in Figure 2, each wavelet at scale 1 covers 2^1 seconds while each wavelet at scale 4 covers $2^4 = 16$ seconds. After removing all the lower scale signals called *detailed signals* from the original signal, we obtain a smoothed version of the original signal called the *approximation signal*. For example, in Figure 2, the original CPU demand signal is decomposed into four detailed signals from scale 1 to 4, and one approximation signal. Then the prediction of the original signal is synthesized by adding up the predictions of these decomposed signals.

Wavelet transforms can use different basis functions such as the Haar and Daubechies wavelets [1]. In contrast, Fourier transforms [6] can only use the sinusoid as the basis function, which only works well for cyclic resource demand traces. Thus, wavelet transforms have advantages over Fourier transforms in analyzing acyclic patterns.

The scale signal i is a series of independent non-overlapping chunks of time, each with duration of 2^i (e.g., the time intervals [0-8), [8-16)). We need to predict $W/2^i$ values to construct the scale i signal in the look-ahead window W as adding one value will increase the length of the scale i signal by 2^i.

Since each wavelet in the higher scale signal has a larger duration, we have fewer values to predict for higher scale signals given the same look-ahead window. Thus, it is easier to achieve accurate predictions for higher scale signals as fewer prediction iterations are needed. For example, in Figure 2, suppose the look-ahead window is 16 seconds, we only need to predict 1 value for the approximation signal but we need to predict 8 values for the scale 1 detail signal.

Wavelet transforms have two key configuration parameters: 1) the wavelet function to use, and 2) the number of scales. AGILE dynamically configures these two parameters in order to minimize the prediction error. Since the approximation signal has fewer values to predict, we want to maximize the similarity between the approximation signal and the original signal. For each sliding window D, AGILE selects the wavelet function

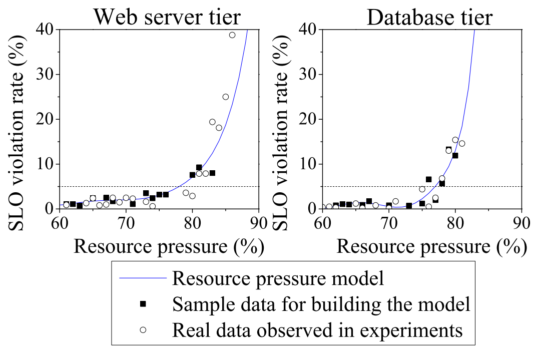


Figure 3: Dynamically derived CPU resource pressure models mapping from the resource pressure level to the SLO violation rate using online profiling for RUBiS web server and database server. The profiling time for constructing one resource pressure model is about 10 to 20 minutes.

that results in the smallest Euclidean distance between the approximation signal and the original signal. Then, AGILE sets the number of values to be predicted for the approximation signal to 1. It does this by choosing the number of scales for the wavelet transforms. Given a look-ahead window W, let U denote the number of scales (e.g., scale of the approximation signal). Then, we have $W/2^U = 1$, or $U = \lceil log_2(W) \rceil$. For example, in Figure 2, the look-ahead window is 16 seconds, so AGILE sets the maximum scale to $U = \lceil log_2(16) \rceil = 4$.

We can use different prediction algorithms for predicting wavelet values at different scales. In our current prototype, we use a simple Markov model based prediction scheme presented in [19].

2.2 Online resource pressure modeling

AGILE needs to pick an appropriate resource allocation to meet the application's SLO. One way to do this would be to predict the input workload [21] and infer the future resource usage by constructing a model that can map input workload (e.g., request rate, request type mix) into the resource requirements to meet an SLO. However, this approach often requires significant knowledge of the application, which is often unavailable in IaaS clouds and might be privacy sensitive, and building an accurate workload-to-resource demand model is nontrivial [22].

Instead, AGILE predicts an application's resource usage, and then uses an application-agnostic *resource pressure* model to map the application's SLO violation rate target (e.g., $< 5\%$) into a maximum resource pressure to maintain. Resource pressure is the ratio of resource usage to allocation. Note that it is necessary to allocate a little more resources than predicted in order to accommodate transient workload spikes and leave some headroom for the application to demonstrate a need for more resources [39, 33, 31]. We use online profiling to derive a resource pressure model for each application tier. For example, Figure 3 shows the relationship between CPU resource pressure and the SLO violation rate for the two tiers in RUBiS, and the model that AGILE fits to the data. If the user requires the SLO violation rate to be no more than 5%, the resource pressure of the web server tier should be kept below 78% and the resource pressure of the database tier below 77%.

The resource pressure model is application specific, and may change at runtime due to variations in the workload mix. For example, in RUBiS, a workload with more write requests may require more CPU than the workload with more browse requests. To deal with both issues, AGILE generates the model dynamically at runtime with an application-agnostic scheme that uses online profiling and curve fitting.

The first step in building a new mapping function is to collect a few pairs of resource pressure and SLO violation rates by adjusting the application's resource allocation (and hence resource pressure) using the Linux `cgroups` interface. If the application consists of multiple tiers, the profiling is performed tier by tier; when one tier is being profiled, the other tiers are allocated sufficient resources to make sure that they are not bottlenecks. If the application's SLO is affected by multiple types of resources (e.g., CPU, memory), we profile each type of resource separately while allocating sufficient amounts of all the other resource types. We average the resource pressures of all the servers in the profiled tier and pair the mean resource pressure with the SLO violation rate collected during a profiling interval (e.g., 1 minute).

AGILE fits the profiling data against a set of polynomials with different orders (from 2 to 16 in our experiment) and selects the best fitting curve using the least-square error. We set the maximum order to 16 to avoid overfitting. At runtime, AGILE continuously monitors the current resource pressure and SLO violation rate, and updates the resource pressure model with the new data. If the mapping function changes significantly (e.g., due to variations in the workload mix), and the approximation error exceeds a pre-defined threshold (e.g., 5%), AGILE replaces the current model with a new one. Since we need to adjust the resource allocation gradually and wait for the application to become stable to get a good model, it takes about 10 to 20 minutes for AGILE to derive a new resource pressure model from scratch using the online profiling scheme. To avoid frequent model retraining, AGILE maintains a set of models and dynamically selects the best model for the current workload. This is useful for applications that have distinct phases of operation. A new model is built and added only if the approximation errors of all current models exceed the threshold.

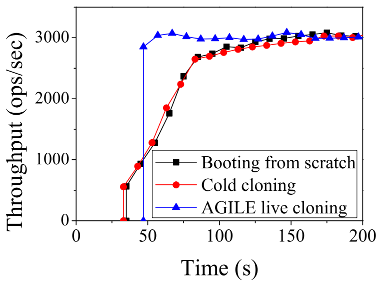


Figure 4: Performance of a new Cassandra server using different server instantiation mechanisms in KVM. All measurements start at the time of receiving a new server cloning request. We expect post-copy live cloning would behave similar to cold cloning.

2.3 Dynamic server pool scaling

Our technique for scaling up the server pool when overload is predicted distinguishes itself from previous work [28, 8] in terms of agility: servers can be dynamically added with little interference, provide near immediate performance scale-up, and low bandwidth cost using adaptive copy rate configuration.

There are multiple approaches to instantiate a new application server:

1. *Boot from scratch*: create a new VM and start the OS and application from the beginning.
2. *Cold cloning*: create a snapshot of the application VM beforehand and then instantiate a new server using the snapshot.
3. *Post-copy live cloning* [28]: instantiate a new server by cloning one of the currently running VMs, start it immediately after instantiation and use demand paging for memory copy.
4. *Pre-copy live cloning*: instantiate a new server from an already running VM. The new server is started after almost all the memory has been copied.

AGILE uses the last of these, augmented with rate control over the data transfer to achieve rapid performance scale-up, minimize interference with the source VMs, and avoid storing and maintaining VM snapshots. Figure 4 shows the throughput of a new Cassandra server [4] using different server instantiation schemes. AGILE allows the new instance to reach its maximum performance immediately, while the others take about 2 minutes to warm up. Note that AGILE triggers the live cloning *before* the application enters the overload state, so its performance is still good during the pre-copy phase, as we will show later.

Our live VM cloning scheme is similar to previous VM/process migration systems [13, 51]. In the pre-copy phase, the dirty memory pages of the source VM are copied iteratively in multiple rounds without stopping the source VM. A stop-and-copy phase, where the source VM is paused temporarily, is used for transferring the remaining dirty pages. A typical pause is within 1 second.

AGILE also performs disk cloning to make the new VM independent of the source VM. In IaaS clouds, the VM's disk is typically located on a networked storage device. Because a full disk image is typically large and would take a long time to copy, AGILE performs *incremental disk cloning* using QEMU Copy On Write (QCOW). When we pause the source VM to perform the final round of memory copy, we make the disk image of the source VM a read-only base image, and build two incremental (copy-on-write) images for the source VM and the new VM. We can associate the new incremental image with the source VM on-the-fly without restarting the VM by redirecting the disk image driver at the hypervisor level. This is transparent to the guest OS of the source VM.

Because live VM cloning makes the new VM instance inherit all the state from the source VM, which includes the IP address, the new VM may immediately send out network packets using the same IP address as the source VM, causing duplicate network packets and application errors. To avoid this, AGILE first disconnects the network interface of the new VM, clears the network buffer, and then reconnects the network interface of the new VM with a new IP address.

AGILE introduces two features to live VM cloning.

Adaptive copy rate configuration. AGILE uses the minimum copy rate that can finish the cloning before the overload is predicted to start (T_o), and adjusts this dynamically based on how much data needs to be transferred. This uses the minimal network bandwidth, and minimizes impact on the source machine and application.

If the new application server configuration takes T_{config} seconds, the cloning must finish within $T_{clone} = T_o - T_{config}$. Intuitively, the total size of transferred memory should equal the original memory size plus the amount of memory that is modified while the cloning is taking place. Suppose the VM is using M memory pages, and the desired copy rate is r_{page_copy} pages per second. We have: $r_{page\_copy} \times T_{clone} = M + r_{dirty} \times T_{clone}$. From this, we have: $r_{page\_copy} = M/T_{clone} + r_{dirty}$. To estimate the page-dirty rate, we continuously sample the actual page-dirtying rate and use an exponential moving average of these values as the estimated value. AGILE will also adjust the copy rate if the predicted overload time T_o changes.

Event-driven application auto-configuration. AGILE allows VMs to subscribe to critical events that occur during the live cloning process to achieve auto-configuration. For example, the new VM can subscribe to the *NetworkConfigured* event so that it can configure itself to use its new IP address. The source VM can subscribe to the *Stopping* event that is triggered when the cloning enters the stop-and-copy phase, so that it can notify a front-end load balancer to buffer some user requests (e.g., write requests). Each VM image is associated with an XML configuration file specifying what to invoke on each cloning event.

Minimizing unhelpful cloning. Since live cloning takes resources, we want to avoid triggering unnecessary cloning on transient workload spikes: AGILE will only trigger cloning if the overload is predicted more than k (e.g. k=3) consecutive times. Similarly, AGILE cancels cloning if the overload is predicted to be gone more than k consecutive times. Furthermore, if the overload state will end before the new VM becomes ready, we should not trigger cloning.

To do this, AGILE checks whether an overload condition will appear in the look ahead window $[t, t+W]$. We want to ignore those transient overload states that will be gone before the cloning can be completed. Let $T_{RML} < W$ denote the required minimum lead time that AGILE's predictor needs to raise an alert in advance for the cloning to complete before the system enters the overload state. AGILE will ignore those overload alarms that only appear in the window $[t, t+T_{RML}]$ but disappear in the window $[t+T_{RML}, t+W]$. Furthermore, cloning is triggered only if the overload state is predicted to last for at least Q seconds in the window $[t+T_{RML}, t+W]$ $(0 < Q \leq W - T_{RML})$.

The least-loaded server in the pool is used as the source VM to be cloned. AGILE also supports concurrent cloning where it creates multiple new servers at the same time. Different source servers are used to avoid overloading any one of them.

Online prediction algorithms can raise false alarms. To address this issue, AGILE continuously checks whether previously predicted overload states still exist. Intuitively, as the system approaches the start of the overload state, the prediction should become more accurate. If the overload state is no longer predicted to occur, the cloning operation will be canceled; if this can be done during the pre-copy phase, it won't affect the application or the source VM.

3 Experimental evaluation

We implemented AGILE on top of the KVM virtualization platform, in which each VM runs as a KVM process. This lets AGILE monitor the VM's resource usage through the Linux `/proc` interface. AGILE periodically samples system-level metrics such as CPU consumption, memory allocation, network traffic, and disk I/O statistics. To implement pre-copy live cloning, we modified KVM to add a new KVM hypervisor module and an interface in the `KVM monitor` that supports starting, stopping a clone, and adjusting the memory copy rate. AGILE controls the resources allocated to application VMs through the Linux `cgroups` interface.

We evaluated our KVM implementation of AGILE using the RUBiS online auction benchmark (PHP version) [38] and the Apache Cassandra key-value store 0.6.13 [4]. We also tested our prediction algorithm using Google cluster data [20]. This section describes our experiments and results.

3.1 Experiment methodology

Our experiments were conducted on a cloud testbed in our lab with 10 nodes. Each cloud node has a quad-core Xeon 2.53GHz processor, 8GiB memory and 1Gbps network bandwidth, and runs 64 bit CentOS 6.2 with KVM 0.12.1.2. Each guest VM runs 64 bit CentOS 5.2 with one virtual CPU core and 2GiB memory. This setup is enough to host our test benchmarks at their maximum workload.

Our experiments on RUBiS focus on the CPU resource, as that appears to be the bottleneck in our setup since all the RUBiS components have low memory consumption. To evaluate AGILE under workloads with realistic time variations, we used one day of per-minute workload intensity observed in 4 different real world web traces [24] to modulate the request rate of the RUBiS benchmark: (1) World Cup 98 web server trace starting at 1998-05-05:00.00; (2) NASA web server trace beginning at 1995-07-01:00.00; (3) EPA web server trace starting at 1995-08-29:23.53; and (4) ClarkNet web server trace beginning at 1995-08-28:00.00. These traces represent realistic load variations over time observed from well-known web sites. The resource usage is collected every 2 seconds. We perform fine-grained sampling for precise resource usage prediction and effective scaling [43]. Although the request rate is changed every minute, the resource usage may still change faster because different types of requests are generated.

At each sampling instant t, the resource demand prediction module uses a sliding window of size D of recent resource usage (i.e., from $t-D$ to t) and predicts future resource demands in the look-ahead window W (i.e., from t to $t+W$). We repeat each experiment 6 times.

We also tested our prediction algorithm using real system resource usage data collected on a Google cluster [20] to evaluate its accuracy on predicting machine overloads. To do this, we extracted CPU and memory usage traces from 100 machines randomly selected from the Google cluster data. We then aggregate the resource usages of all the tasks running on a given machine to get the usage for that machine. These traces represent various realistic workload patterns. The sampling interval in the Google cluster is 5 minutes and the trace lasts 29 days.

Parameter	RUBiS	Google data
Input data window (D)	6000 seconds	250 hours
Look-ahead window (W)	120 seconds	5 hours
Sampling interval (T_s)	2 seconds	5 minutes
Total trace length	one day	29 days
Overload duration threshold (Q)	20 seconds	25 minutes
Response time SLO	100 ms	NA

Table 1: Summary of parameter values used in our experiments.

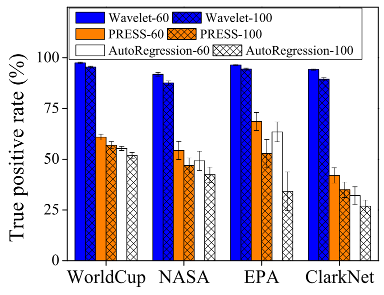
Table 1 shows the parameter values used in our experiments. We also performed comparisons under different threshold values by varying D, W, and Q, which show similar trends. Note that we used consistently larger D, W, and Q values for the Google trace data because the sampling interval of the Google data (5 minutes) is significantly larger than what we used in the RUBiS experiments (2 seconds).

To evaluate the accuracy of our wavelet-based prediction scheme, we compare it against the best alternatives we could find: PRESS [19] and auto-regression [9]. These have been shown to achieve higher accuracy and lower overheads than other alternatives. We calculate the overload-prediction accuracy as follows. The predictor is deemed to raise a valid overload alarm if the overload state (e.g., when the resource pressure is bigger than the overload threshold) is predicted earlier than the required minimum lead time (T_{RML}). Otherwise, we call the prediction a false negative. Note that we only consider those overload states that last at least Q seconds (Section 2.3). Moreover, we require that the prediction model accurately estimates when the overload will start, so we compare the predicted alarm time with the true overload start time to calculate a *prediction time error*. If the absolute prediction time error is small (i.e., $\leq 3 \cdot T_s$), we say the predictor raises a correct alarm. Otherwise, we say the predictor raises a false alarm.

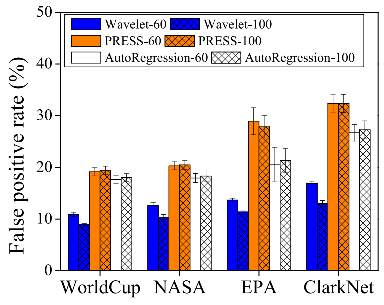
We use the standard metrics, *true positive rate* (A_T) and *false positive rate* (A_F), given in equation 1. P_{true}, P_{false}, N_{true}, and N_{false} denote the number of true positives, false positives, true negatives, and false negatives, respectively.

$$A_T = \frac{P_{true}}{P_{true}+N_{false}}, \quad A_F = \frac{P_{false}}{P_{false}+N_{true}} \qquad (1)$$

A service provider can either rely on the application itself or an external tool [5] to tell whether the application SLO is being violated. In our experiments, we adopted the latter approach. With the RUBiS benchmark, the workload generator tracks the response time of the HTTP requests it makes. The SLO violation rate is the fraction



(a)



(b)

Figure 5: CPU demand prediction accuracy comparison for RUBiS web server driven by one-day request traces of different real web servers with T_{RML} = 60 and 100 seconds.

of requests that have response time larger than a predefined SLO threshold. In our experiments, this was 100ms, the 99th percentile of observed response times for a run with no resource constraints. We conduct our RUBiS experiments on both the Apache web server tier and the MySQL database tier.

For comparison, we also implemented a set of alternative resource provisioning schemes:

- *No scaling*: A non-elastic resource provisioning scheme that cannot change the size of the server pool, which is fixed at 1 server as this is sufficient for the average resource demand.
- *Reactive*: This scheme triggers live VM cloning when it observes that the application has become overloaded. It uses a fixed memory-copy rate, and for a fair comparison, we set this to the average copy rate used by AGILE so that both schemes incur a similar network cost for cloning.
- *PRESS*: Instead of using the wavelet-based prediction algorithm, PRESS uses a Markov+FFT resource demand prediction algorithm [19] to predict future overload state and triggers live cloning when an overload state is predicted to occur. PRESS uses the same false alarm filtering mechanism described in Section 2.3.
- *FixThreshold-65% and -80%*: This scheme triggers live VM cloning if the resource pressure exceeds 65% and 80%. This allows us to evaluate the effects of the resource pressure model.

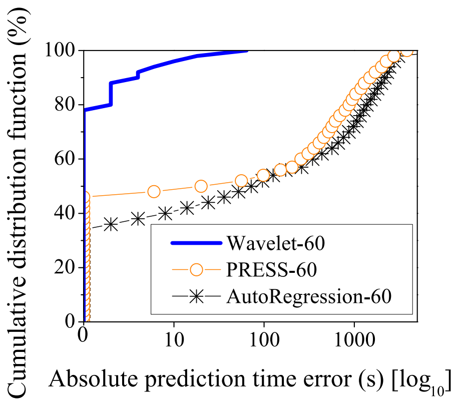


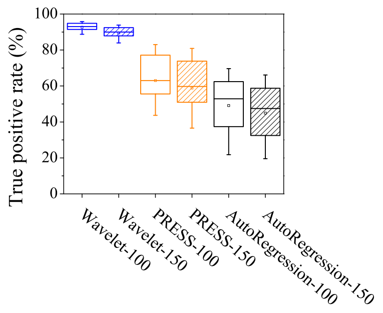
Figure 6: Cumulative distribution function of the prediction time error for the RUBiS web server driven by the ClarkNet workload.

Note that the *reactive* and *PRESS* schemes use the AGILE same resource pressure model to decide the resource pressure threshold for the target 5% SLO violation rate.

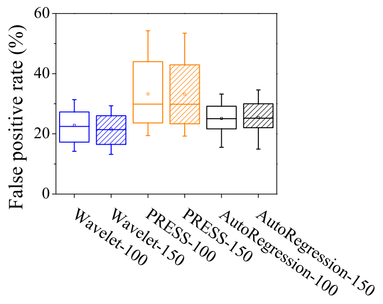
3.2 Experimental results

Prediction accuracy results. In this set of experiments, no cloning is performed. Figure 5 shows the overload prediction accuracy comparisons for RUBiS driven by different real workload traces. We test the prediction system with different lead time requirements (T_{RML}). The results show that our wavelet prediction scheme is statistically significantly better than the PRESS scheme and the auto-regression scheme (the independent two-sample t-test indicates *p-value* ≤ 0.01). Particularly, the wavelet scheme can improve the true positive rate by up to 3.42$\times$ and reduce the false positive rate by up to 0.41$\times$. The accuracy of the PRESS and auto-regression schemes suffers as the number of iterations increases, errors accumulate, and the correlation between the prediction model and the actual resource demand becomes weaker. This is especially so for ClarkNet, the most dynamic of the four traces.

In the above prediction accuracy figure, we consider the predictor raises a correct alarm if the absolute prediction time error is less than $\leq 3 \cdot T_s$. We further compare the distributions of the absolute prediction time error among different schemes. Figure 6 compares the cumulative distribution functions of the absolute prediction time error among different schemes. We observe that AGILE achieves much lower prediction time error (78% alarms have 0 absolute prediction time

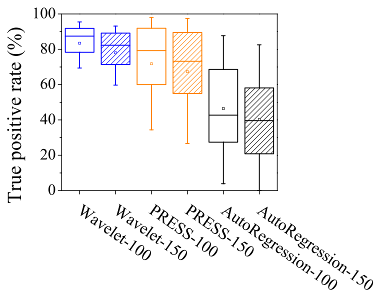


(a)

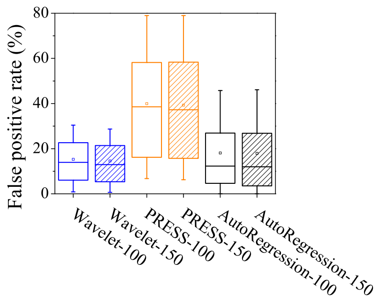


(b)

Figure 7: Prediction accuracy for 100 Google cluster CPU traces with T_{RML} = 100 and 150 minutes. The bottom and top of the box represent 25th and 75th percentile values, the ends of the whiskers represent 10th and 90th percentile values.



(a)



(b)

Figure 8: Prediction accuracy comparison for 100 Google cluster memory traces.

error) than auto-regression (34% alarms have 0 absolute prediction time error) and PRESS (46% alarms have 0 absolute prediction time error). Other traces show similar trend, which are omitted due to space limitation.

Figure 7 and Figure 8 show the prediction accuracy for the CPU and memory usage traces on 100 machines in a Google cluster. The overload threshold is set to the 70th percentile of all values in each trace. We observe that the wavelet scheme again consistently outperforms the PRESS scheme and the auto-regression scheme with up to 2.1× better true positive rate and 0.34× the false positive rate.

Overload handling results. Next, we evaluate how well AGILE handles overload using dynamic server pool scaling. The experiment covers 7000 seconds of a RUBiS run driven by the ClarkNet web server trace. The first 6000 seconds are used for training and no cloning is performed. The overload state starts at about t = 6500s. When examining the effects of scaling on different tiers in RUBiS, we limit the scaling to one tier and allocate sufficient resources to the other tier. We repeat each experiment 3 times.

Figure 9 shows the overall results of different schemes. Overall SLO violation *rate* denotes the percentage of requests that have response times larger than the SLO violation threshold (e.g., 100ms) during the experiment run. SLO violation *time* is the total time in which SLO violation rate (collected every 5 seconds) exceeds the target (e.g., 5%). We observe that AGILE consistently achieves the lowest SLO violation rate and shortest SLO violation time. Under the *no scaling* scheme, the application suffers from high SLO violation rate and long SLO violation time in both the web server tier and the database tier scaling experiments. The *reactive* scheme mitigates this by triggering live cloning to create a new server after the overload condition is detected, but since the application is already overloaded when the scaling is triggered, the application still experiences a high SLO violation rate for a significant time. The *FixThreshold-80%* scheme triggers the scaling too late, especially in the database experiment and thus does not show any noticeable improvement compared to without scaling. Using a lower threshold, *FixThreshold-65%* improves the SLO violation rate but at a higher resource cost: resource pressure is maintained at 65% while AGILE maintains the resource pressure at 75%. In contrast, AGILE predicts the overload state in advance, and successfully completes live cloning before the application enters the overload state. With more accurate predictions, AGILE also outperforms PRESS by predicting the overload sooner.

Figure 10 shows detailed performance measurements

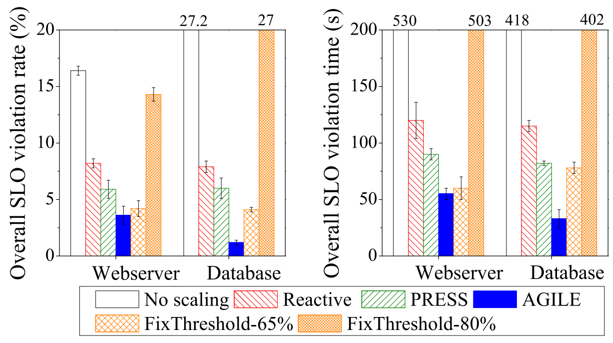


Figure 9: SLO violation rates and times for the two RUBiS tiers under a workload following the ClarkNet trace.

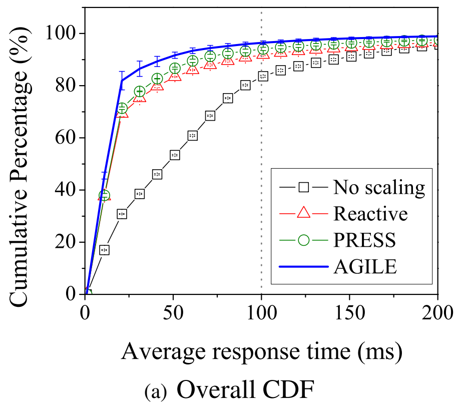
Application	In use	Copied	Ratio
RUBiS Webserver	530MiB	690MiB	1.3×
RUBiS Database	1092MiB	1331MiB	1.2×
Cassandra	671MiB	1001MiB	1.5×

Table 2: Amount of memory moved during cloning for different applications.

for the web server tier during the above experiment. We sample the average response time every second and plot the cumulative distribution functions for the whole run and during cloning. From Figure 10(a), we can see that the response time for most requests meets the SLO when using the AGILE system. In contrast, if no scaling is performed, the application suffers from a significant increase in response time. Figure 10(b) shows that all the scaling schemes, except AGILE, cause much worse performance during the cloning process: the application is overloaded and many requests suffer from a large response time until a new server is started. In contrast, using AGILE, the application experiences little response time increase since the application has not yet entered the overload state. Figure 11 shows the equivalent results for the database server and has similar trends.

Figure 12 and Figure 13 show the SLO violation rate timeline of RUBiS application under the ClarkNet workload. Compared to other schemes, AGILE triggers scaling before the system enters the overload state. Under the reactive scheme, the live cloning is executed when the system is already overloaded, which causes a significant impact to the application performance during the cloning time. Although PRESS can predict the overload state in advance, the lead time is not long enough for cloning to finish before the application is overloaded.

Dynamic copy-rate configuration results. Table 2 shows the amount of memory moved during cloning for different applications. AGILE moved at most 1.5 times the amount of the memory in use at the source VM. We also tested AGILE under different overload pending



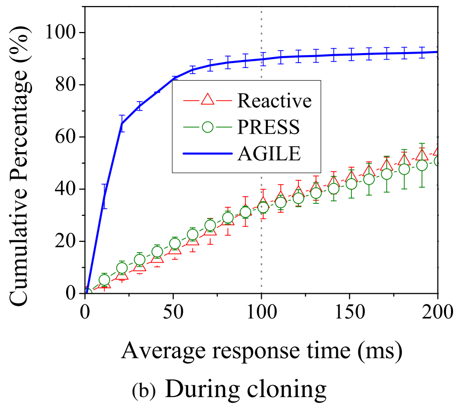


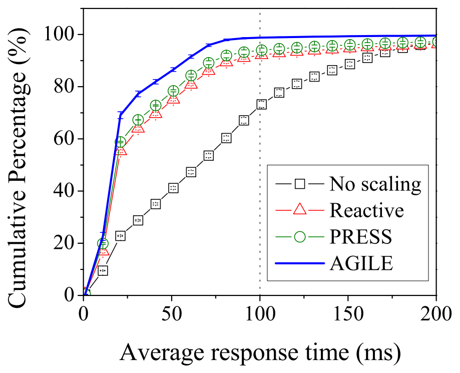
Figure 10: Scaling up the RUBiS web server tier from 1 server to 2 servers under a dynamic workload following the ClarkNet trace. *(a) Overall CDF* denotes the whole experiment. *(b) During cloning* denotes the period in which the scaling is being executed. AGILE always triggers scaling earlier than other schemes.

time deadlines (i.e., target time to finish cloning) and check whether the cloning can finish within the pending time. Figure 14 shows that our dynamic copy-rate setting can accurately control the cloning time under different deadlines.

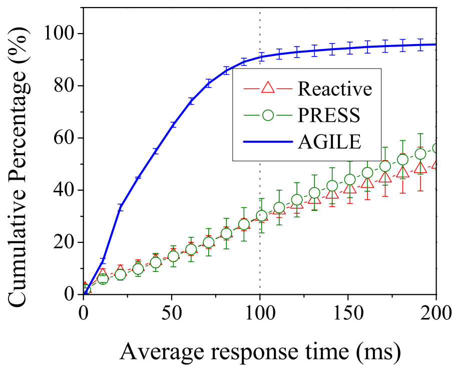
We measured the time spent in the different stages of the live VM cloning for different applications (Table 3). As expected, pre-copy dominates the cloning time (tens of seconds), while the stop-and-copy time is only 0.1 s, so the downtime of the source VM is negligible.

Overhead results. We first present the overhead imposed by our online profiling mechanism. Figure 15 shows the timeline of the average response time during profiling. Figure 16 shows the performance impact of the online profiling on the average response time over the period of 6 hours, in which AGILE performs profiling three times. Overall, the overhead measurements show that AGILE is practical for online system management.

We also evaluated the overhead of the AGILE system. The AGILE slave process on each cloud node imposes



(a) Overall CDF



(b) During cloning

Figure 11: Scaling up the RUBiS database server tier from 1 server to 2 servers under a dynamic workload following the ClarkNet trace. We used 9 web servers to make the database tier become the bottleneck.

Application	Pre-copy	Stop-and-copy	Configuration
RUBiS Webserver	31.2 ± 1.1 s	0.10 ± 0.01 s	16.8 ± 0.6 s
RUBiS Database	33.1 ± 0.9 s	0.10 ± 0.01 s	17.8 ± 0.8 s
Cassandra	31.5 ± 1.1 s	0.10 ± 0.01 s	17.5 ± 0.9 s

Table 3: Time spent in the different stages of live VM cloning.

less than 1% CPU overhead. The most computationally intensive component is the prediction module that runs on the master node. Table 4 shows the online training time and prediction time for AGILE, PRESS, and auto-regression schemes. AGILE has similar overheads at the master node as does PRESS. The auto-regression scheme is faster, however its accuracy is much worse than AGILE. Clearly, these costs still need to be reduced (e.g., by incremental retraining mechanisms and decentralized masters), and we hope to work on this in the future.

4 Related Work

AGILE is built on top of previous work on resource demand prediction, performance modeling, and VM cloning. Most previous work on server pool scaling (e.g., [29, 17]) adopts a *reactive* approach while AGILE provides a *prediction-driven* solution that allows the system to start new instances before SLO violation occurs.

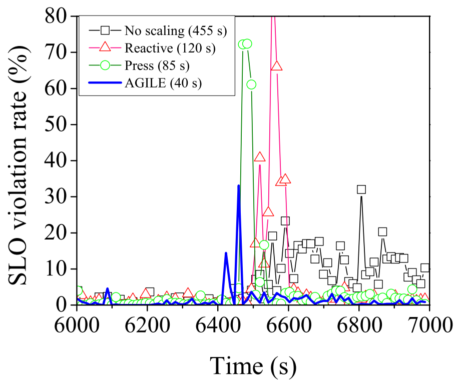


Figure 12: SLO violation timeline for web server tier experiment under the ClarkNet workload. The number in the bracket indicates the SLO violation time in seconds.

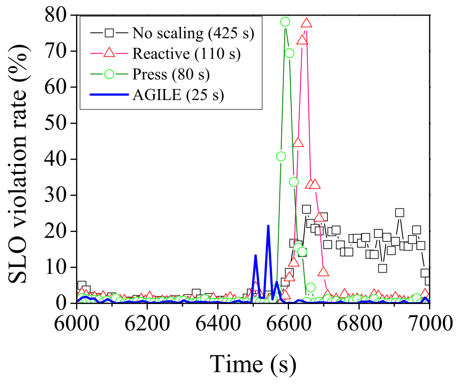


Figure 13: SLO violation timeline for database tier experiment under the ClarkNet workload.

Previous work has proposed white-box or grey-box approaches to addressing the problem of cluster sizing. Elastisizer [22] combines job profiling, black-box and white-box models, and simulation to compute an optimal cluster size for a specific MapReduce job. Verma et al. [47] proposed a MapReduce resource sizing framework that profiles the application on a smaller data set and applies linear regression scaling rules to generate a set of resource provisioning plans. The SCADS director framework [44] used a model-predictive control (MPC) framework to make cluster sizing decisions based on the current workload state, current data layout, and predicted SLO violation. Huber et al. [23] presented a self-adaptive resource management algorithm which leverages workload prediction and a performance model [7] that predicts application's performance

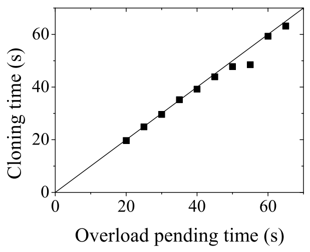


Figure 14: Cloning time achieved against predicted time to overload.

Scheme	Training time (3000 samples)	Prediction time (60 steps)
AGILE	575 ± 7 ms	2.2 ± 0.1 ms
PRESS	595 ± 6 ms	1.5 ± 0.1 ms
Auto-regression	168 ± 5 ms	2.2 ± 0.1 ms

Table 4: Prediction model training time and the prediction time comparison between AGILE, PRESS, and auto-regression schemes. The prediction module runs on the master host.

under different configurations and workloads. In contrast, AGILE does not require any prior application knowledge.

Previous work [53, 26, 35, 36, 34, 29] has applied control theory to achieve adaptive resource allocation. Such approaches often have parameters that need to be specified or tuned offline for different applications or workloads. The feedback control system also requires a feedback signal that is stable and well correlated with SLO measurement. Choosing suitable feedback signals for different applications is a non-trivial task [29]. Other projects used statistical learning methods [41, 42, 15, 40] or queueing theory [46, 45, 14] to estimate the impact of different resource allocation policies. Overdriver [48] used offline profiling to learn the memory overload probability of each VM to select different mitigation strategies: using migration for sustained overloads or network memory for transient overloads. Those models need to be built and calibrated in advance. Moreover, the resource allocation system needs to make certain assumptions about the application and the running platform (e.g., input data size, cache size, processor speed), which often is impractical in a virtualized, multi-tenant IaaS cloud system.

Trace-driven resource demand prediction has been applied to several dynamic resource allocation problems. Rolia et al. [37] described a resource demand prediction scheme that multiplies recent resource usage by a burst factor to provide some headroom. Chandra et al. [11] developed a prediction framework based on auto-regression to drive dynamic resource allocation decisions. Gmach et al. [18] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. Andrzejak et al. [3] employed a

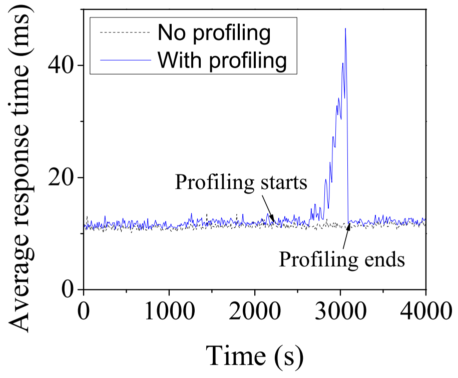


Figure 15: The effect of profiling on average response time for the RUBiS system under the ClarkNet workload.

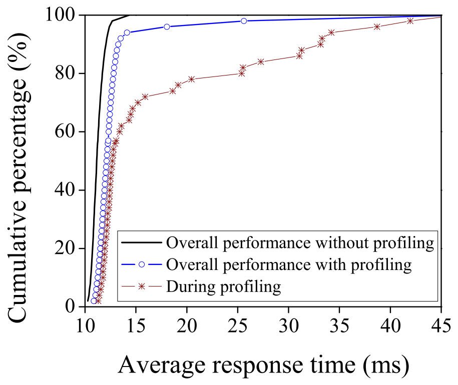


Figure 16: Profiling overhead for the RUBiS system under the ClarkNet workload. Profiling occurs every two hours.

genetic algorithm and fuzzy logic to address the problem of having little training data. Gandhi et al. [16] combined long-term predictive provisioning using periodic patterns with short-term reactive provisioning to minimize SLO violations and energy consumption. Matsunaga et al. [30] investigated several machine learning techniques for predicting spatio-temporal resource utilization. PRESS [19] developed a hybrid online resource demand prediction model that combines a Markov model and a fast Fourier transform-based technique. Previous prediction schemes either focus on short-term prediction or need to assume cyclic workload patterns. In contrast, AGILE focuses on medium-term prediction and works for arbitrary workload patterns.

VM cloning has been used to support elastic cloud computing. SnowFlock [28] provides a fast VM instantiation scheme using on-demand paging. However, the new instance suffers from an extended performance warmup period while the working set is copied over from the origin. Kaleidoscope [8] uses fractional VM cloning with VM state coloring to prefetch semantically-related regions. Although our current prototype uses full pre-copy, AGILE could readily work with fractional pre-copy too: prediction-driven live cloning and dynamic copy rate adjustment can be applied to both cases. Fractional pre-copy could be especially useful if the overload duration is predicted to be short. Dolly [10] proposed a proactive database provisioning scheme that creates a new database instance in advance from a disk image snapshot and replays the transaction log to bring the new instance to the latest state. However, Dolly did not provide any performance predictions, and the new instance created from an image snapshot may need some warmup time. In contrast, the new instance created by AGILE can reach its peak performance immediately after start.

Local resource scaling (e.g., [39]) or live VM migration [13, 50, 49, 25] can also relieve local, per-server application overloads, but distributed resource scaling will be needed if the workload exceeds the maximum capacity of any single server. Although previous work [39, 50] has used overload prediction to proactively trigger local resource scaling or live VM migration, AGILE addresses the specific challenges of using predictions in distributed resource scaling. Compared to local resource scaling and migration, cloning requires longer lead time and is more sensitive to prediction accuracy, since we need to pay the cost of maintaining extra servers. AGILE provides medium-term predictions to tackle this challenge.

5 Future Work

Although AGILE showed its practicality and efficiency in experiments, there are several limitations which we plan to address in our future work.

AGILE currently derives resource pressure models for just CPU. Our future work will extend the resource pressure model to consider other resources such as memory, network bandwidth, and disk I/O. There are two ways to build a multi-resource model. We can build one resource pressure model for each resource separately or build a single resource pressure model incorporating all of them. We plan to explore both approaches and compare them.

AGILE currently uses resource capping (a Linux `cgroups` feature) to achieve performance isolation among different VMs [39]. Although we observed that the resource capping scheme works well for common bottleneck resources such as CPU and memory, there may still exist interference among co-located VMs [52]. We need to take such interference into account to build more precise resource pressure models and achieve more accurate overload predictions.

Our resource pressure model profiling can be triggered either periodically or by workload mix changes. To make AGILE more intelligent, we plan to incorporate workload change detection mechanism [32, 12] in AGILE. Upon detecting a workload change, AGILE starts a new profiling phase to build a new resource pressure model for the current workload type.

6 Conclusion

AGILE is an application-agnostic, prediction-driven, distributed resource scaling system for IaaS clouds. It uses wavelets to provide medium-term performance predictions; it provides an automatically-determined model of how an application's performance relates to the resources it has available; and it implements a way of cloning VMs that minimizes application startup time. Together, these allow AGILE to predict performance problems far enough in advance that they can be avoided.

To minimize the impact of cloning a VM, AGILE copies memory at a rate that completes the clone just before the new VM is needed. AGILE performs continuous prediction validation to detect false alarms and cancels unnecessary cloning.

We implemented AGILE on top of the KVM virtualization platform, and conducted experiments under a number of time-varying application loads derived from real-life web workload traces and real resource usage traces. Our results show that AGILE can significantly reduce SLO violations when compared to existing resource scaling schemes. Finally, AGILE is lightweight, which makes it practical for IaaS clouds.

7 Acknowledgement

This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] N. A. Ali and R. H. Paul. *Multiresolution signal decomposition*. Academic Press, 2000.

[2] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[3] A. Andrzejak, S. Graupner, and S. Plantikow. Predicting resource demand in dynamic utility computing environments. In *Autonomic and Autonomous Systems*, 2006.

[4] Apache Cassandra Database. `http://cassandra.apache.org/`.

[5] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

[6] E. Brigham and R. Morrow. The fast Fourier transform. *IEEE Spectrum*, 1967.

[7] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Automated Software Engineering*, 2011.

[8] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *EuroSys*, 2011.

[9] E. S. Buneci and D. A. Reed. Analysis of application heartbeats: Learning structural and temporal features in time series data for identification of performance problems. In *Supercomputing*, 2008.

[10] E. Cecchet, R. Singh, U. Sharma, and P. Shenoy. Dolly: virtualization-driven database provisioning for the cloud. In *VEE*, 2011.

[11] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *IWQoS*, 2003.

[12] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks*, 2008.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[14] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[15] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: better decisions enabled by machine learning. In *International Conference on Data Engineering*, 2009.

[16] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Green Computing Conference and Workshops*, 2011.

[17] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. In *Transactions on Computer Systems*, 2012.

[18] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *International Conference on Web Services*, 2007.

[19] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *International Conference on Network and Service Management*, 2010.

[20] Google cluster-usage traces: format + scheme (2011.11.08 external). `http://goo.gl/5uJri`.

[21] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *International Conference on Performance Engineering*, 2013.

[22] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[23] N. Huber, F. Brosig, and S. Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Software Engineering for Adaptive and Self-Managing Systems*, 2011.

[24] The IRCache Project. `http://www.ircache.net/`.

[25] C. Isci, J. Liu, B. Abali, J. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. In *IBM Journal of Research and Development*, 2011.

[26] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *ICAC*, 2009.

[27] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Linux Symposium*, 2007.

[28] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[29] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ICAC*, 2010.

[30] A. Matsunaga and J. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Cluster, Cloud and Grid Computing*, 2010.

[31] A. Neogi, V. R. Somisetty, and C. Nero. Optimizing the cloud infrastructure: tool design and a case study. *International IBM Cloud Academy Conference*, 2012.

[32] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. FChain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[33] Oracle. Best practices for database consolidation in private clouds, 2012. `http://www.oracle.com/technetwork/database/focus-areas/database-cloud/database-cons-best-practices-1561461.pdf`.

[34] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated

control of multiple virtualized resources. In *EuroSys*, 2009.

[35] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[36] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Real-Time Systems*, 2002.

[37] J. Rolia, L. Cherkasova, M. Arlitt, and V. Machiraju. Supporting application quality of service in shared resource pools. *Communications of the ACM*, 2006.

[38] RUBiS Online Auction System. `http://rubis.ow2.org/`.

[39] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*, 2011.

[40] P. Shivam, S. Babu, and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *VLDB*, 2006.

[41] P. Shivam, S. Babu, and J. S. Chase. Learning application models for utility resource planning. In *ICAC*, 2006.

[42] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: cross-platform management for internet services. In *USENIX ATC*, 2008.

[43] Y. Tan, V. Venkatesh, and X. Gu. Resilient self-compressive monitoring for large-scale hosting infrastructures. In *TPDS*, 2012.

[44] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. In *FAST*, 2011.

[45] B. Urgaonkar and A. Chandra. Dynamic provisioning of multi-tier internet applications. In *ICAC*, 2005.

[46] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS*, 2005.

[47] A. Verma, L. Cherkasova, and R. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *Middleware*, 2011.

[48] D. Williams, H. Jamjoom, Y. Liu, and H. Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *VEE*, 2011.

[49] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: virtualize once, run everywhere. In *Eurosys*, 2012.

[50] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[51] E. Zayas. Attacking the process migration bottleneck. In *SOSP*, 1987.

[52] X. Zhang, E. Tune, R. Hagmann, R. J. V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Eurosys*, 2013.

[53] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 Islands: integrated capacity and workload management for the next generation data center. In *ICAC*, 2008.

PACMan: Performance Aware Virtual Machine Consolidation

Alan Roytman
University of California, Los Angeles

Aman Kansal
Microsoft Research

Sriram Govindan
Microsoft Corporation

Jie Liu
Microsoft Research

Suman Nath
Microsoft Research

Abstract

Consolidation of multiple workloads, encapsulated in virtual machines (VMs), can significantly improve efficiency in cloud infrastructures. But consolidation also introduces contention in shared resources such as the memory hierarchy, leading to degraded VM performance. To avoid such degradation, the current practice is to not pack VMs tightly and leave a large fraction of server resource unused. This is wasteful. We present a system that consolidates VMs such that performance degradation is within a tunable bound while minimizing unused resources. The problem of selecting the most suitable VM combinations is **NP**-Complete and our system employs a practical method that performs provably close to the optimal. In some scenarios resource efficiency may trump performance and for this case our system implements a technique that maximizes performance while not leaving any resource unused. Experimental results show that the proposed system realizes over 30% savings in energy costs and up to 52% reduction in performance degradation compared to consolidation algorithms that do not consider degradation.

1 Introduction

Average server utilization in many data centers is low, estimated between 5% and 15% [10]. This is wasteful because an idle server often consumes more than 50% of its peak power [11], implying that servers at low utilization consume significantly more energy than fewer servers at high utilization. Additionally, low utilization implies a greater number of servers being used, resulting in wasted capital. One solution to prevent such wastage is to *consolidate* applications on fewer servers.

Consolidation inevitably introduces resource contention resulting in performance degradation. To mitigate this contention, data centers virtualize resources and split them across applications consolidated on shared hardware. However, virtualization does not prevent all forms of contention and hence does not completely eliminate performance degradation. In particular, contention in shared caches and memory bandwidth degrades performance significantly, as measured for a variety of workloads [3–5, 16, 17, 19, 21, 32, 35]. Execution times increase by several tens of percent.

To reduce degradation, prior works have measured the degradations for possible VM combinations and then co-locate those VMs that lead to the least degradation [17, 18, 29]. But this approach does not respect a target performance bound. Performance is often paramount for Internet services. Measurements on Amazon, Microsoft and Google services show that a fraction of a second increase in latency can result in revenue losses as high as 1% to 20% [13, 20, 26]. A knee-jerk reaction then is to forgo all or part of the savings from consolidation. In Google data centers for instance, consolidated workloads use only 50% of the processor cores [21]. Every other processor core is left unused simply to ensure that performance does not degrade.

We wish to preserve the performance of consolidated VMs, but *not waste excessive resources in doing so.* The challenges are to (1) determine how much each VM will degrade when placed with different sets of VMs to be consolidated, and (2) identify *which* and *how many* VMs can be placed on a server such that required performance is maintained. The problem of identifying suitable VMs turns out to be **NP**-Complete, and we design a computationally efficient algorithm that we prove performs close to the theoretical optimal. As a result, the excess resources left unused in our approach are significantly lower than in current practice.

An additional mechanism to preserve performance after consolidation is to improve the isolation of resources in hardware [3, 5, 16, 28, 35], or software [1, 4, 6, 27, 32]. Further, excess resources may be allocated at run time [23] to overcome degradation. These approaches are complementary because they do not determine the best VMs to be placed together in the first place. Our method can make that determination, and then these techniques can be applied with a lower overhead.

Specifically, we make the following contributions:

First, we present a *p*erformance *a*ware *c*onsolidation *man*ager, PACMan, that minimizes resource cost, such as energy consumption or number of servers used. PACMan consolidates VMs such that performance degradation stays within a specified bound. Since this problem is **NP**-complete, PACMan uses an approximate but computationally efficient algorithm that we prove performs logarithmically close to the optimal.

Second, while customer-facing applications prioritize performance, batch processes, such as Map-Reduce [8], may prioritize resource efficiency. For such situations PACMan provides an "Eco" mode, that fills up all server cores, and minimizes worst case degradation. We specifically consider worst case, as opposed to average considered in [17], since in Map-Reduce, reduce cannot start until all map tasks have completed and hence, only the degradation of the worst hit map task matters. We show that it is difficult to design provably near-optimal methods for this scenario and present a suitable heuristic.

Finally, we evaluate PACMan using degradations measured on SPEC CPU 2006 applications. For minimizing wasted resource while preserving performance, PACMan operates within about 10% of the optimal, saves over 30% energy compared to consolidation schemes that do not account for interference, and improves total cost of operations by 22% compared to current practice. For the Eco mode, PACMan yields up to 52% reduction in degradation compared to naïve methods.

2 PACMan Design

This section describes the performance repercussion of consolidation and how our design addresses it.

2.1 Problem Description

Consolidation typically relies on virtual machines (VMs) for resource and fault isolation. Each VM is allocated a fixed share of the server's resources, such as a certain number of cores on a multi-core server, a certain fraction of the available memory, storage space, and so on. In theory, each VM should behave as if it is a separate server: software crashes or resource bottlenecks in one VM should not affect other VMs on the same server. In practice however, VM resource isolation is not perfect. Indeed, CPU cores or time slices, memory space, and disk space can be isolated well using existing virtualization products, and methods have emerged for other resources such as network and storage bandwidth [22, 34]. However, there remain resources, such as *shared caches and memory bandwidth*, that are hard to isolate. Hence, consolidated applications, even when encapsulated in VMs, may suffer resource contention or *interference*, and this leads to performance degradation.

Example: Consider a toy data center with 4 VMs, A, B, C, and D (Figure 1). On the left, the 4 VMs are placed on a single server each. Suppose the task inside each VM takes 1 hour to finish. The shaded portion of the vertical bars represents the energy used over an hour; the darker rectangle represents the energy used due to the server being powered on (idle power consumption) and the rectangles labeled with the VM name represent the additional energy consumed in VM execution (increase in server energy due to processor resource use). On the right, these VMs are consolidated on two servers (the other two are in sleep mode).

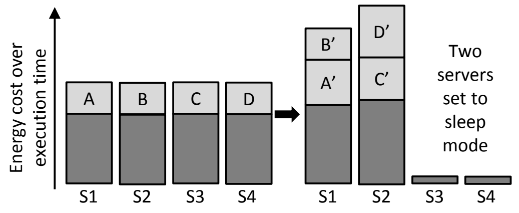

Energy cost over execution time
A
B
C
D
S1
S2
S3
S4
B'
A'
D'
C'
Two servers set to sleep mode
S1
S2
S3
S4

Figure 1: Energy cost change due to consolidation.

The setup on the right is more efficient. However, due to resource contention, the execution time goes up for most of the jobs. Both the server idle energy and the additional energy used by each job increase due to the longer run time. The increase in energy consumption due to contention may wipe out some or all of the energy savings obtained by turning off two servers. Also, longer running time may violate quality of service (QoS) requirements.

One may minimize performance degradation by placing each VM in a separate server, but that obviously reduces efficiency. On the other hand, one may maximize efficiency by packing the VMs into the minimum number of servers required to satisfy the number of processor cores, memory and disk space requirements of each VM, but such packing hurts performance.

2.2 System Overview

Our goal is to select the right set of VMs to co-locate on each server such that performance constraints are satisfied and wasted resource is minimized.

2.2.1 Assumptions

We make three assumptions about the system:

Degradation: We assume that the performance degradation suffered by each VM, when consolidated with any set of other VMs, is known from existing methods [12, 19, 21]. These methods do not require explicit performance measurement for each possible set of VMs. Rather, a VM is profiled individually to generate an *interference profile*. Profiling takes a few hundred milliseconds depending on the cache architecture. These profiles can be used to compute the expected degradation for any set of VMs placed together. Small errors in prediction can be addressed by including an error margin in the performance bound, and consolidating to within that conservative bound. Explicit measurement may also be used for a small number of VMs, as in [17]. We focus on the consolidation method given the degradations. Since our algorithms work given any interference data, the techniques we use can be applied to cross-socket interference or any other type of interference as well, so long as it can be quantified and measured.

Temporal Demand Variations. We assume that as demand for an application varies, the number of VM instances hosting that app are increased or decreased to match the demand. Overloading a small number of VMs would degrade performance while leaving VMs underutilized would incur excess cost to host them. Hence, commercial tools are available to dynamically change the number of VMs [24, 33]. This implies that the degradation data or interference profile needs to be collected only for the desired demand level, rather than at multiple demand levels that a VM may serve. If demand is lower than that served by a single VM instance for the application, we conservatively use the profile at its optimal demand level.

VM to Processor Core Mapping: We assume that each VM is assigned one core, following the model in [3, 12, 16, 17, 19, 32]. Considering VMs that span multiple cores does not change the problem fundamentally. However, if multiple VMs share a single core, the nature of resource contention may change, and existing degradation estimation methods [12, 19, 21] will not suffice. If alternate degradation modeling methods are available or explicit measurements of degradations are provided, our consolidation algorithm would extend to that case.

2.2.2 Architecture

The PACMan system architecture is shown in Figure 2. The system consists of the following three components:

Conservatively Packed Servers: Customers submit VMs through appropriate cloud APIs. Ideally, a VM placement solution should determine the optimal placement for each VM as soon as it arrives, such that the entire set of VMs currently running in the cloud is optimally placed. However, since such an optimal online solution is not available, we focus on a *batched* operating scenario. The cloud initially hosts the incoming VMs on

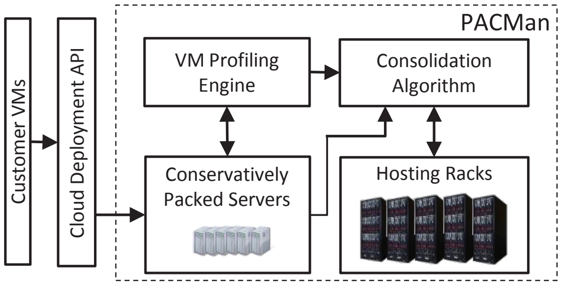


Figure 2: PACMan block diagram.

conservatively packed servers, for a batching period (say 30 to 60 minutes). These servers may comprise a small fraction (say 1%-5%) of the data center. Conservative placement implies that a significant amount of resources are left unused to avoid interference, such as by leaving alternate processor cores empty [21]. Since the VM is active, it does not matter to the customer that it is placed on a conservatively packed server.

VM Profiling Engine: While a VM is running on the conservatively packed servers, profiling methods from [12, 21] are applied to the VMs[1]. These methods characterize a VM while it is running normally, and generate a set of parameters that allow estimating the performance degradation that will be *suffered* and *caused* by the VM when consolidated with other VMs. Their prediction accuracy is high (5-10% of actual performance), as measured on real data center and benchmark applications. Given n VMs and k core servers, only $O(n)$ measurements are needed, even though the number of possible consolidated sets is $O(n^k)$.

Consolidation Algorithm: At the end of each batching period, PACMan uses the VM consolidation algorithm proposed in this paper to place the VMs on *hosting racks* that comprise the bulk of the cloud's infrastructure. Most of the data center thus operates efficiently using the near-optimal placement. The inputs to the algorithm are the VM interference characteristics obtained by the profiling engine. The output is a placement of VMs that respects performance constraints and minimizes unused resources. Typically, other algorithms (including bin packing methods such as best fit or first fit) do not take interference into account, and hence cannot consolidate VMs efficiently. The design of PACMan algorithms is presented in the next two sections.

[1]We use [12] in our prototype. In this method, each VM is mapped to a *clone* application, which closely mimics the application's interference signature. A discrete set of clones covers the entire spectrum of memory-subsystem interference behaviors. Thus, a potentially unbounded number of applications are mapped to a finite number of clones. A one-time profiling step maps a new VM to a known clone. The clones are then used as a proxy for predicting performance for different consolidation sets.

3 Performance Mode

The first mode of PACMan operation, denoted the performance mode (P-mode), determines the best sets and their sizes such that performance constraints are not violated. It may leave some processor cores unused, unlike prior methods that use up every core [17, 18] but may violate performance constraints.

Servers and VMs: Suppose m chip-multiprocessors (CMPs) are available, each with k cores. We are primarily interested in the inter-core interference within a CMP. The VMs placed on the same CMP suffer from this degradation. If a server happens to have multiple processor sockets, we assume there is no interference among those. As a result, multiple CMPs within a server may be treated independently of each other. We loosely refer to each CMP as a separate server as shown in Figure 3. We are given n VMs to be placed on the above servers, such that each VM is assigned one core.

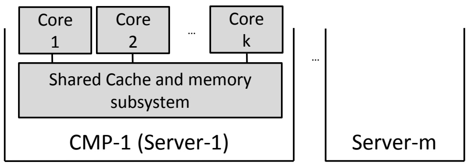


Figure 3: CMPs (referred to as servers) with k cores. Contention in the shared cache and memory hierarchy degrades the performance of VMs in the same server.

Degradation: Suppose that the set of VMs placed together on a server are denoted by S. For singleton sets, i.e., a VM j running alone, there is no degradation and we denote this using a degradation $d_j = 1$. For larger sets, the degradation for VM $j \in S$ is denoted by $d_j^S \geq 1$. For example, for two co-located VMs, $S = \{A, B\}$, suppose A's running time increases by 50% when it runs with B, relative to when it runs alone, while B is unaffected by A. Then, $d_A^S = 1.5$ and $d_B^S = 1$.

We assume that adding more VMs to a set may only increase (or leave unchanged) the degradation of previously added VMs.

3.1 Consolidation Goal

The consolidation objective may be stated as follows.

P-Mode: (Minimize resource cost subject to a performance constraint)

Given

n VMs,

Servers with k cores,

Degradations for all sets of VMs up to size k,

Cost $w(S)$ for a set of VMs S placed on a server, and

Maximum tolerable degradation $D \geq 1$ for any VM[2].

Find a placement of the n VMs using some number, b, of servers, to minimize

$$\sum_{i=1}^{b} w(S_i)$$

where S_i represents the set of VMs placed on the i^{th} server.

Cost Metric: The resource cost, $w(S)$, to be minimized may represent the most relevant cost to the system. For instance, if we wish to minimize the number of servers used, then we could use $w(S) = 1$ for any set S regardless of how many VMs S contains. To minimize energy, $w(S)$ could be defined as the sum of a fixed cost c_f and a dynamic cost c_d. The fixed cost c_f models the idle energy used for keeping a server active, and may also include capital expense. The dynamic cost, c_d, models the increase in energy due to VMs assigned to the server. For concreteness, we consider the cost function $w(S)$ to be the energy cost. The specific values used for c_f and c_d are described in Section 5 along with the evaluations. Our solution is applicable to any cost function that monotonically increases with the number of VMs.

Batched Operation: The problem above assumed all VMs are given upfront. In practice, following the setup from Figure 2, only the VMs that arrived in the most recent batching period will be consolidated. Each batch will hence be placed optimally using P-mode consolidation, but the overall placement across multiple batches may be sub-optimal. Hence, once a day, such as during times of low demand, the placement solution can be jointly applied to all previously placed VMs, and the placement migrated to the jointly optimal placement. The joint placement satisfies the same performance constraints but may reduce resource cost even further.

3.1.1 Problem Complexity

The complexity is different depending on whether the servers have only $k = 2$ cores or more than 2 cores.

Dual-Core servers: For $k = 2$ cores, there is a polynomial time algorithm that can compute the optimal solution. The main idea is to construct a weighted, undirected graph on $2n$ nodes. The first n nodes represent the VMs, and the others are "dummy" nodes (one for each VM). For VM pairs whose degradation is below the bound D, we place an edge connecting them and assign an edge weight equal to the cost of placing those two VMs together. We place an edge between each VM node and its dummy node with a weight that corresponds to the cost

[2]We assume that the performance constraint is the same for all VMs though multiple quality of service classes, each with their own degradation limit, could be considered as well and do not fundamentally change the problem.

of running that VM alone. Finally, we place edges of weight 0 between all pairs of dummy nodes. Finding the best pairs of VMs for a consolidated placement is equivalent to computing a minimum cost perfect matching on this graph. Graph algorithms are available to compute a minimum cost perfect matching in polynomial time. We omit details for this case since most data center servers have more than 2 cores.

NP-Completeness: For servers with more than two cores ($k \geq 3$), the problem is **NP**-Complete. This is because it can be thought of as a variant of the k-Set Cover problem. In the k-Set Cover problem, we have a universe U of elements to cover (each element could represent a VM), along with a collection C of subsets each of size at most k (the subsets could represent sets of VMs with degradation below D). Placing VMs on servers corresponds to finding the minimum number of disjoint VM subsets that cover all VMs. Assuming $w(S) = 1$ for all sets S, the k-Set Cover problem becomes a special case of the P-mode problem, i.e., solving the P-mode problem enables solving the k-Set Cover problem. The k-Set Cover problem is **NP**-Complete [9]. Hence, the P-mode problem is **NP**-Complete.

3.2 Consolidation Algorithm

Since the problem is **NP**-Complete for $k \geq 3$ cores, we propose a computationally efficient algorithm that finds a near-optimal placement.

Using the profiling method described in Section 2.2, it is easy to filter out VM sets that violate the degradation constraint. Suppose the collection of remaining sets (VM combinations that can be used) is denoted by $\mathscr{F}$.

First, for each set $S \in \mathscr{F}$, the algorithm assigns a value $V(S) = w(S)/|S|$. Intuitively, this metric characterizes the cost of a set S of VMs. Sets with more VMs (larger set size, $|S|$) and low resource use ($w(S)$) yield low $V(S)$.

Second, the algorithm sorts these sets in ascending order by $V(S)$. Sets that appear earlier in the ascending order have lower cost.

The final step is to make a single pass through this sorted list, and include a set S as a placement in the consolidation output if and only if it is disjoint from all sets that have been chosen earlier. The algorithm stops after it has made a single pass through the list. The algorithm can stop earlier if all the VMs are included in the chosen sets. The first set in the sorted list will always be taken to be in the solution since nothing has been chosen before it and it is hence disjoint. If the second set is disjoint from the first set, then the algorithm takes it in the solution. If the second set has at least one VM in common with the first, the algorithm moves onto the third set, and so on. The precise specification is given in Algorithm 1.

Example: Consider a toy example with three VMs, A, B, and C and $k = 2$ cores. Suppose the characterization from the VM profiling engine results in the degradation numbers shown in Table 1. Suppose the performance constraint given is that no VM should degrade more than 10% ($D = 1.1$) and the cost metric $w(S)$ is just the number of servers for simplicity ($w(S) = 1$ for any set). A set with two VMs ($|S| = 2$) will have $V(S) = 1/2$ while a set with one VM will have $V(S) = 1$. Then filtering out the sets that cause any of the VMs to have a degradation greater than D, and computing the $V(S)$ metric for each set, we get the sorted list as: BC, AB, A, B, C. The algorithm first selects set BC and allocates it to a server (VMs B and C thus share a single server). The next set AB is not disjoint from BC and the algorithm moves to the subsequent set A. This is disjoint and is allocated to another server. All VMs are now allocated and the algorithm stops.

Algorithm 1 CONSOLIDATE($\mathscr{F}$, n, k, D)

```
1: Compute $V(S) \leftarrow \frac{w(S)}{|S|}$, for all $S \in \mathscr{F}$
2: $\mathscr{L} \leftarrow$ Sorted sets in $\mathscr{F}$ such that $V(S_i) \leq V(S_j)$ if $i \leq j$
3: $\mathbb{L} \leftarrow \phi$
4: for $i = 1$ to $|\mathscr{L}|$ do
5:     if $S_i$ is disjoint from every set in $\mathbb{L}$ then
6:         $\mathbb{L} \leftarrow \mathbb{L} \cup \{S\}$
7: Return $\mathbb{L}$
```

VM Set	AB	AC	BC	A	B	C
d_{VM}^{Set}	$d_A = 1.1$ $d_B = 1.1$	$d_A = 1.0$ $d_C = 1.5$	$d_B = 1.0$ $d_C = 1.1$	1	1	1

Table 1: Degradations for VMs in the example.

Complexity: The algorithm operates in polynomial time since sorting is a polynomial time operation, $O(|\mathscr{F}| \cdot log(|\mathscr{F}|))$. The subsequent step requiring a single pass through the list has linear time complexity. At every step in the linear pass the algorithm needs to check if each VM in the set being selected has been assigned already and this can be achieved in constant time as follows. Maintain a boolean bit-vector for every VM indicating if it has been assigned yet. For the set being checked, just look up this array, which takes at most $O(k)$ time per set since the set cannot have more than k VMs. Also, after selecting a set we update the boolean array, which again takes constant time.

While the computation time is polynomial in the size of the input, the size of the input can be large. The list of degradation values for all possible VM sets has size $|\mathscr{F}| = O(n^k)$ elements, which can be large for a cloud infrastructure hosting thousands of VMs. However, when the degradation estimation technique from [12] is used, all VMs are mapped to a finite set of clones and the number of clones does not grow with the number of VMs. We can treat all VMs that map to a common clone as one type of VM. The number of clones used to map all VMs then represents the distinct types of VMs in the input. For instance, for the characterization technique in [12], for quad-core servers, at most 128 types of clones are required, and not all of them may be used for a particular set of input VMs.

Suppose the n VMs can be classified into $\tau \leq 128$ types. Then, the algorithm only needs to consider all sets S from τ VM types with possibly repeated set elements. The number of these sets is $O(\tau^k)$, which is manageable in practice since τ does not grow very large, even when n is large.

The algorithm changes slightly to accommodate multiple VMs of each type. The assignment of value $V(S)$ and the sorting step proceed as before. However, when doing the single pass over the sorted list, when a disjoint set S is chosen, it is repeatedly allocated to servers as long as there is at least one unallocated instance of each VM type required for S. The resultant modification to Algorithm 1 is that $\mathscr{F}_\tau$ is provided as input instead of $\mathscr{F}$ where $\mathscr{F}_\tau$ denotes the collection of all feasible sets of VM types with repeated elements, and at step 5, instead of checking if the VMs are not previously allocated, one repeats this step while additional unallocated VMs of each type in the set remain.

Correctness: The algorithm always assigns every VM to a server since all singleton sets are allowed and do appear in the sorted list (typically after the sets with large cardinality). Also, it never assigns a VM to more than one server since it only picks disjoint sets, or sets with unallocated VM instances when VM-types are used, while making the pass through the sorted list. Hence, the algorithm always obtains a correct solution.

3.3 Solution Optimality

A salient feature of this algorithm is that the consolidation solution it generates is guaranteed to be near-optimal, in terms of the resources used.

Let ALG denote the allocated sets output by the proposed algorithm, and let OPT be the sets output by the optimal solution. Define the resource cost of the proposed algorithm's solution to be $E(ALG)$, and that of the optimal algorithm as $E(OPT)$. We will show that for *every* possible collection of VMs to be consolidated,

$$E(ALG) \leq H_k \cdot E(OPT)$$

where H_k is the k^{th}-Harmonic number. $H_k = \sum_{i=1}^{k} \frac{1}{i} \approx \ln(k)$.

In other words, the resource cost of the solution generated by the proposed algorithm is within $\ln(k)$ of the resource cost of the optimal solution. Given that k is constant for a data center and does not increase with the number of VMs, this is a very desirable accuracy guarantee. The proof is inspired by the approximation quality proof for the weighted k-Set Cover problem [7, 15]. However, we cannot pick overlapping sets (since choosing sets in our setting corresponds to choosing a placement of VMs onto servers), and the input sets are closed under subsets.

Theorem 1. *For all inputs, the proposed algorithm outputs a solution that is within a factor $H_k \approx \ln(k)$ of the optimal solution.*

Proof. By definition, we have

$$E(ALG) = \sum_{S \in ALG} w(S).$$

Assign a cost to each VM $c(j)$ as follows: whenever the proposed algorithm chooses a set S to be part of its solution, set the cost of each VM $j \in S$ to be $c(j) = w(S)/|S|$ (these costs are only for analysis purposes, the actual algorithm never uses $c(j)$). Hence,

$$E(ALG) = \sum_{S \in ALG} |S| \frac{w(S)}{|S|} = \sum_{S \in ALG} \sum_{j \in S} c(j) = \sum_{j=1}^{n} c(j),$$

where the last equality holds because the set of VMs in the solution is the same as all VMs given in the input. Then, since the optimal solution also assigns all VMs to servers:

$$E(ALG) = \sum_{j=1}^{n} c(j) = \sum_{S^* \in OPT} \sum_{j \in S^*} c(j),$$

where $S^* \in OPT$ is a set chosen by the optimal solution. Suppose, for the moment, we could prove that the last summation term above satisfies $\sum_{j \in S^*} c(j) \leq H_k w(S^*)$. Then we would have

$$E(ALG) \leq \sum_{S^* \in OPT} H_k w(S^*) = H_k \cdot E(OPT).$$

All we have left to prove is that, for any $S^* \in OPT$, we indeed have $\sum_{j \in S^*} c(j) \leq H_k w(S^*)$. Consider any set S^* in the optimal solution and order the VMs in the set according to the order in which the *proposed* algorithm covers the VMs, so that $S^* = \{j_s, j_{s-1}, \ldots, j_1\}$. Here, j_s is the first VM from S^* to be covered by the proposed algorithm, while j_1 is the last VM to be covered by the proposed algorithm in potentially different sets. In case the proposed algorithm chooses a set which covers several VMs from S^*, we just order these VMs arbitrarily.

Now, consider VM $j_i \in S^*$ immediately before the proposed algorithm covers it with a set T. At this time, there are at least i VMs which are not covered, namely

$j_i, j_{i-1}, \ldots, j_1$. There could be more uncovered VMs in S^*, for instance, if the proposed algorithm chose set T such that T covers VMs $j_s, \ldots, j_i$, then all VMs in S^* would be considered uncovered immediately before set T is chosen. Moreover, since the optimal solution chose S^*, and since sets are closed under subsets, it must be the case that the proposed algorithm could have chosen the set $S = \{j_i, \ldots, j_1\}$ (since it is a feasible set and it is disjoint). At each step, since the proposed algorithm chooses the disjoint set T that minimizes $w(T)/|T|$, it must be the case that $w(T)/|T| \leq w(S)/|S|$. By our assumption that energy costs can only increase if VMs are added, we have $w(S) \leq w(S^*)$, and hence VM j_i is assigned a cost of $w(T)/|T| \leq w(S)/|S| \leq w(S^*)/|S| = w(S^*)/i$. Summing over all costs of VMs in S^*, we have

$$\sum_{j \in S^*} c(j) \leq \sum_{j_i \in S^*} w(S^*)/i = H_s \cdot w(S^*) \leq H_k \cdot w(S^*)$$

(since $|S^*| = s \leq k$). Hence, $\sum_{j \in S^*} c(j) \leq H_k \cdot w(S^*)$ indeed holds and this completes the proof. □

To summarize, we provide a polynomial time algorithm that is guaranteed to provide a solution within a logarithmic factor of the optimal. Note that this is a worst-case guarantee, and in practice we can expect the solution quality to be better (e.g., our experimental results in Section 5). In fact, our approximation guarantee is asymptotically the best approximation factor one could hope for, due to the hardness of approximation lower bound known for the k-Set Cover problem [30] (hence, there are worst-case instances in which any algorithm must perform poorly, but these instances typically do not occur in practice and the algorithms perform much better).

4 Eco-Mode

In some cases, such as batch based data processing, resource efficiency may take precedence over performance. For such scenarios, PACMan provides a resource efficient mode of operation, referred to as the Eco-mode. Here, the number of servers used is fixed and the VMs are tightly packed. The goal is to minimize the degradation. Prior works have minimized average degradation [17, 18, 29] and their heuristics can be used in Eco-mode. We additionally consider worst case degradation. The worst case degradation is especially important for parallel computing scenarios where the end result is obtained only after all parallelized tasks complete and hence performance is bottle-necked by the worst hit VM.

Eco-Mode: (Minimize maximum degradation)
Given
n VMs,
m servers with k cores ($n \leq mk$), and
Degradations for all sets of VMs up to size k,
Find an allocation of the n VMs to m servers which minimizes the objective

$$\max_{1 \leq i \leq m} \max_{j \in S_i} d_j^{S_i}$$

where S_i represents the set of VMs placed on the i^{th} server ($|S_i| \leq k$ for each i).

As in the previous case, while the 2-core case can be solved in polynomial time, the Eco-mode problem becomes **NP**-Complete for $k \geq 3$.

Efficient Near-Optimal Algorithms: Given that the problem is **NP**-Complete, a polynomial time algorithm to compute the optimal solution is unlikely to be found, unless $\mathbf{P} = \mathbf{NP}$. The next best thing would be an efficient algorithm that computes a provably near-optimal solution.

Surprisingly, for $k \geq 3$ a computationally efficient algorithm that guarantees the solution to be within any polynomial factor of the optimal cannot be found. For instance, a computationally efficient algorithm that can guarantee its solution to be within a factor n^{100} of the optimal cannot be found.

Theorem 2. *For the Eco-mode consolidation problem, it is* **NP***-Hard to approximate the optimal solution to within any factor that is polynomial in the number of VMs and servers.*

The proof is relegated to a tech report [25] for brevity.

Algorithm: The implication of the above theorem is that any computationally efficient Eco-mode algorithm will have to rely on heuristics.

The heuristic we propose greedily improves a given placement using VM swaps. A swap refers to exchanging the placement of one VM with another. Start out with any initial placement of VMs. Consider all possible placements that are reachable from the existing placement in a single swap. In each such placement, for each server, compute the degradation of the worst hit VM on that server, using the degradation characterization from the VM profiling engine. Take the sum of these worst case degradations on all servers as the cost of that VM placement.

Among all possible placements reachable within a swap, greedily select the one with the lowest cost and actually perform the swap required to reach that placement. Repeat the above process as long as additional swaps are allowed or until a swap no longer yields an improvement.

The work of [31] studies the cost of swapping, giving insight into the trade-off between improving resource efficiency and swapping VMs. To this end, we limit the number of swaps allowed to terminate the search at $G = (k-1)(m-1)$. Starting with an arbitrary placement, it is possible to reach *any* other placement (e.g., the optimal placement) by performing at most $G = (k-1)(m-1)$ swaps. This holds because for each server, we can imagine one of the VMs to be in the correct position on that server, and hence there can be at most $k-1$ VMs on that server that are out of place. By swapping two VMs, we can assign each VM which is on the wrong server to the right server. Hence, each server can be fixed in at most $k-1$ swaps. Once $m-1$ servers have been fixed, the last server must already be correct. However, determining the appropriate number of swaps is not easy and our heuristic is not guaranteed to find the optimal placement in $G = (k-1)(m-1)$ swaps, or any number of swaps. Hence, the number of allowed swaps may be set based on other constraints such as limits on tolerable swap overheads or a threshold on minimum improvement expected from a swap.

While not provably near-optimal, our heuristic is beneficial to the extent that it improves performance compared to naïve methods.

5 Experimental Results

In this section, we quantify the resource savings and performance advantages of using PACMan consolidation for realistic scenarios. Ideally, we wish to compare the practical algorithm used in PACMan with the theoretical optimal, but the optimal is not feasible to compute (these problems are **NP**-Complete) except for very small input sizes. Hence, we illustrate the performance of the proposed methods with respect to the optimal for a few small input instances ($n = 16$ VMs, $m \geq \lceil n/k \rceil$). For more realistic inputs, relevant to real data centers (10^3 VMs), we compare the performance to naïve methods that are unaware of the performance degradation and with one current practice that leaves alternate processor cores unused [21]. For these cases, we also compute the degradation overhead compared to a hypothetical case where resource contention does not cause any degradation. This comparison shows an upper bound on how much further improvement one could hope to make over the PACMan methods.

5.1 Experimental Setup

Degradation Data: We use measured degradation data for SPEC CPU 2006 benchmark applications. These degradations are in the same range as measured for Google's data center workloads in [21], which includes batch and interactive workloads. Since the degradation data is representative of both interactive workloads and batch workloads, it is relevant for both P-mode and Eco-mode.

In particular, we select 4 of the SPEC CPU benchmark applications for which we have detailed interference data for all possible combinations, namely: `lbm`, `soplex`, `povray`, and `sjeng` (some combinations shown in Table 2). These span a range of interference values from low to high. When experimenting with n VMs, we generate an equal number, $n/4$, of each. We do not vary VM degradations over time during VM execution.

Application VMs (S_i)	**Degradations (%)**
`lbm, soplex`	2, 19.7
`soplex, soplex`	10, 10
`lbm, soplex, sjeng`	2, 10, 4.1
`lbm, povray, lbm`	19.6, 5.32, 19.6
`lbm, soplex` `soplex, sjeng`	14.56, 36.9, 36.9, 5.83
`lbm, lbm, lbm, lbm`	104.6 (each)

Table 2: Sample degradation data for the application VMs used in experiments. Degradations are measured on a quad-core processor. For combinations with only 2 or 3 VMs, the remaining cores are unused. Degradations over 100% imply that the execution time of the workload increases by more than twice.

Cloud Configuration: We assume that each server has $k = 4$ cores since quad-core servers are commonly in use. While a server may have many cores across multiple processor sockets, the relevant value of k is the number of cores sharing the same cache hierarchy, since that is where most of the interference occurs. Using real world degradation data, we simulate our proposed algorithm for the cloud configuration described above.

5.2 Performance Mode

The P-mode problem optimizes resource cost given a degradation constraint. The evaluation metric of interest is thus the resource cost. We choose energy as our resource metric. Each server has a fixed and dynamic energy component (Section 3), resulting in an energy cost $w(S) = c_f + \sum_{j \in S} d_j^S$. Here, the additional cost of each VM is being modeled as d_j^S. Considering that running 4 VMs each with an incremental cost of 1 or more would add an additional 4 units of dynamic resource cost, we set the fixed cost $c_f = 4$ to reflect about 50% of the total server energy as the fixed idle cost, which is representative of current server technology. Newer generation servers are trending towards better energy proportionality and idle power costs as low as 30% are expected in the near future. A lower idle power cost will only exaggerate the fraction of overhead due to interference and lead to even greater savings in PACMan.

Comparison with Optimal: To facilitate computation of the optimal, we use a small number, 16, of VMs, with equal proportion of VMs from each of the four benchmarks. We vary the degradation constraint from 10% ($D = 1.1$), to as high as 50%[3]. Aside from the optimal, we also compare against a naïve method that does not quantitatively manage degradation but conservatively leaves every other core unused [21].

Figure 4 shows the energy overhead of the consolidation determined by PACMan, and by the naïve method, over and above the energy used by the optimal method. The proposed approach is within 10% of the optimal, and is significantly better than the naïve approach currently in use.

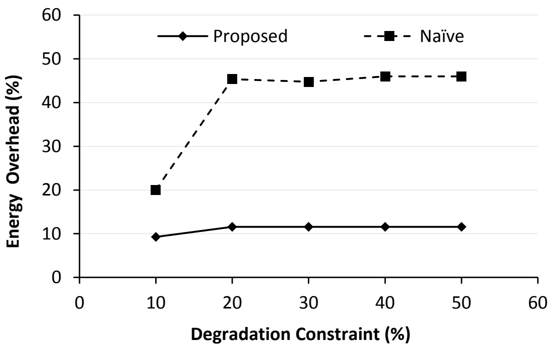


Figure 4: (P-Mode) Excess energy used compared to that used by the optimal solution (computable for a small number of VMs).

Figure 5 shows the server utilizations achieved by the three methods at equivalent performance. The proposed method achieves over 80% utilization in most cases yielding good resource use. Of course, when the degradation allowed is small, servers must be left under-utilized to avoid interference, and even the optimal method cannot use all cores.

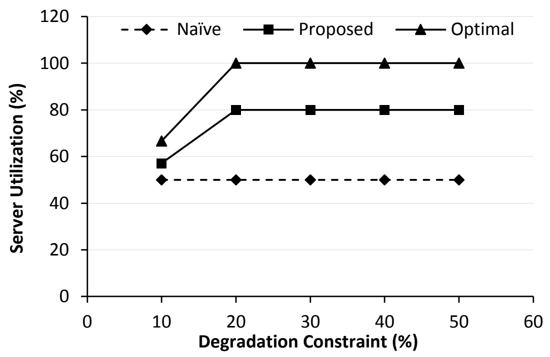


Figure 5: (P-Mode) Server utilizations achieved by the theoretical optimal, proposed, and naïve algorithms.

[3]We start at 10% instead of 0%, since if no degradation is allowed, most VMs would require a dedicated machine, with no benefits from consolidation.

Large number of VMs: The second set of experiments uses more realistic input sizes, up to $n = 1000$ VMs, again taking an equal proportion of VMs from each of the four SPEC CPU applications listed in Section 5.1. Since it is not feasible to compute the optimal solution for a large number of VMs, we compare against a lower bound: resources used when interference has no effect. In reality, interference will lead to a nonzero overhead and the optimal should be expected to be somewhere between 0% and the overhead seen for the proposed method. Figure 6 shows the results, with a performance constraint of 50% ($D = 1.5$), for varying n. We see that the proposed method performs significantly better than the naïve one.

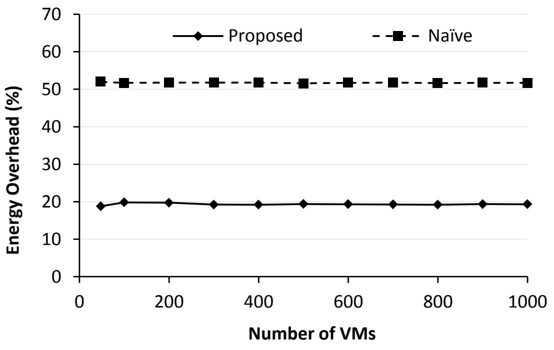


Figure 6: (P-Mode, Large number of VMs) Resource overhead comparison, normalized with respect to hypothetical resource use when there is no interference, which is a lower bound on the optimal.

5.3 Eco-Mode

For the Eco-mode problem, we again compute the optimal solution for a small set $n = 16$ VMs with $m = 4$ servers, with the VMs taken from the SPEC CPU benchmarks. The initial allocation of VMs to servers is arbitrary and we repeat the experiment 10 times, starting with a random initial allocation each time. Since *any* allocation can be reached in at most $(k-1)(m-1) = 9$ swaps, we vary the number of allowed swaps G from 2 to 9. As an additional point of comparison we use a naïve approach that does not consider interference and places the VMs randomly. The performance of the randomized approach is averaged across 10 trials.

Figure 7 shows the excess degradation suffered by the VMs compared to that in the optimal allocation. The practical heuristic used in PACMan performs very closely to the optimal and has up to 30% lower degradation than the naïve method.

Next we vary the number of VMs up to $n = 1000$, packed tightly on $m = n/4$ quad core servers. The ap-

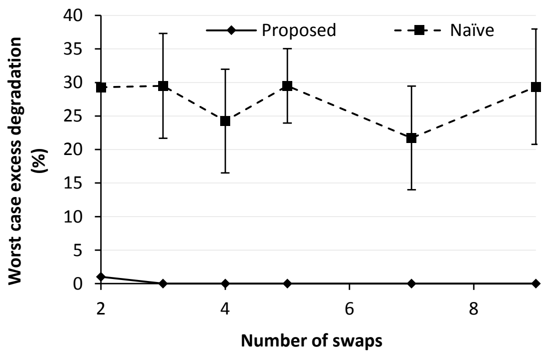


Figure 7: (Eco-mode) Comparison of proposed heuristic and a naïve random algorithm with the theoretical optimal (computable for small input instances). Excess worst case degradation compared to that in the optimal solution is shown. The error bars show the standard deviation across 10 random runs for the naïve approach.

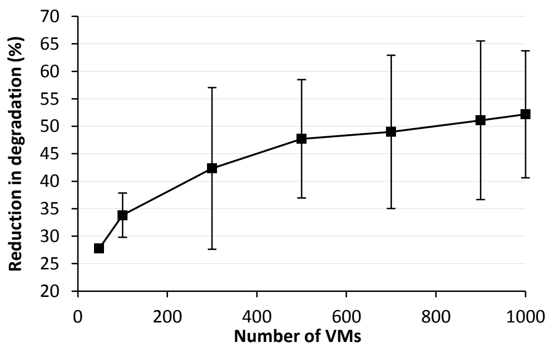


Figure 8: (Eco-mode, Large number of VMs) Reduction in degradation compared to a naïve approach. The error bars show the standard deviation across 10 random placements for the naïve approach.

plications are taken from the SPEC CPU benchmarks as before, in equal proportion. The naïve approach used for comparison is a random placement that does not account for interference (10 random trials are performed for each point).

Since it is not feasible to compute the optimal solution, we use the naïve approach as the base case and show the reduction in degradation achieved by PACMan (Figure 8). The worst case degradation is reduced by 27% to 52% over the range of the number of VMs. While the number of servers is a fixed constraint, reduction in performance degradation results in a corresponding increase in throughput or reduction in runtime, yielding a proportional saving in energy per unit work performed.

In summary, we see that PACMan performs well on realistic degradation data.

5.4 TCO Analysis

The total cost of ownership (TCO) of a data center includes both the operating expenses such as energy bills paid based on usage, and capital expenses, paid upfront. Consolidation affects multiple components of TCO. The resultant savings in TCO are described below.

To compare capital costs and operating expenses using a common metric, James Hamilton provided an amortized cost calculation of an entire data center on a monthly basis [14]. In this calculation, the fixed costs are amortized over the life of the component purchased. For instance, building costs are amortized over fifteen years while server costs are amortized over three years. This converts the capital costs into a monthly expense, similar to the operating expense.

Figure 9 shows the savings resulting in various data center cost components. The baseline we use is the current practice of leaving alternate cores unused [21], and we compare this with our proposed performance preserving consolidation method. In all, a 22% reduction in TCO is achieved, which for a 10MW data center implies that the monthly operating expense is reduced from USD 2.8 million to USD 2.2 million.

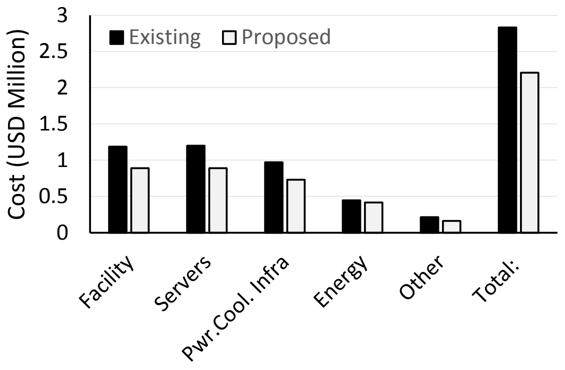


Figure 9: (P-Mode) TCO reduction using the proposed performance preserving consolidation method. Pwr. Cool. Infra. refers to the power and cooling infrastructure cost, as defined in [14].

6 Related Work

Performance isolation from memory subsystem interference has been studied at different levels of the system stack: the hardware level [3, 5, 16, 28, 35], the OS/software level [1, 6, 23, 27], and the VM scheduler level [2, 4, 32]. Our method is complementary in that we facilitate determining the placements with lower interference to which above isolation techniques can then be applied, and are likely to be more effective.

Performance estimates due to interference [12, 19, 21] have also been developed to aid VM placement. We build upon the above works and use the interference characterization provided by them to determine the placements with lower interference.

Consolidation methods taking interference into account have been studied in [17], along with variants for unequal job lengths [29]. Another method to model VM interference through cache co-locality, and a heuristic for run-time scheduling to minimize degradation, was presented in [18]. We allow optimizing for a different objective. While heuristics were proposed in prior works, we provide an algorithm with provable guarantees on the solution quality. We also provide a new inapproximability result.

7 Conclusions

VM consolidation is one of the key mechanisms to improve data center efficiency, but can lead to performance degradation. We presented consolidation mechanisms that can preserve performance.

The extent of performance degradation depends on both how many VMs are consolidated together on a server and which VMs are placed together. Hence, it is important to intelligently choose the best combinations. For many cases, performance is paramount and consolidation will be performed only to the extent that it does not degrade performance beyond the QoS guarantees required for the hosted applications. We presented a system that consolidated VMs within performance constraints. While the problem of determining the best suited VM combinations is **NP**-Complete, we proposed a polynomial time algorithm which yields a solution provably close to the optimal. In fact, the solution was shown to be within $\ln(k)$ of the optimal where k is the number of cores in each server, and is independent of the number of VMs, n. This is a very tight bound for practical purposes. We also considered the dual scenario where resource efficiency is prioritized over performance. For this case, we showed that even near-optimal algorithms with polynomial time complexity are unlikely to be found. Experimental evaluations showed that the proposed system performed well on realistic VM performance degradations, yielding over 30% savings in energy and up to 52% reduction in degradation.

We believe that the understanding of performance aware consolidation developed above will enable better workload consolidation. Additional open problems remain to be addressed in this space and further work is required to develop consolidation methods that operate in an online manner and place VMs near-optimally as and when they arrive for deployment.

References

[1] AWASTHI, M., SUDAN, K., BALASUBRAMONIAN, R., AND CARTER, J. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (2009).

[2] BAE, C. S., XIA, L., DINDA, P., AND LANGE, J. Dynamic adaptive virtual core mapping to improve power, energy, and performance in multi-socket multicores. In *Proceedings of the twenty-first International Symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 247–258.

[3] BITIRGEN, R., IPEK, E., AND MARTINEZ, J. F. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the forty-first annual IEEE/ACM International Symposium on Microarchitecture* (2008).

[4] BOBROFF, N., KOCHUT, A., AND BEATY, K. Dynamic placement of virtual machines for managing sla violations. In *Proceedings of the International Symposium on Integrated Network Management* (2007).

[5] CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (2005).

[6] CHO, S., AND JIN, L. Managing distributed, shared L2 caches through os-level page allocation. In *Proceedings of the thirty-ninth annual IEEE/ACM International Symposium on Microarchitecture* (2006).

[7] CHVATAL, V. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research 4* (1979), 233–235.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: A flexible data processing tool. *Communications of the ACM 53*, 1 (2010), 72–77.

[9] DUH, R., AND FÜRER, M. Approximation of k-set cover by semi-local optimization. In *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing* (1997).

[10] ENVIRONMENTAL PROTECTION AGENCY, U. S. Report to congress on server and data center energy efficiency public law 109-431. Tech. rep., EPA ENERGY STAR Program, 2007.

[11] GANDHI, A., HARCHOL-BALTER, M., DAS, R., AND LEFURGY, C. Optimal power allocation in server farms. In *Proceedings of ACM SIGMETRICS* (2009).

[12] GOVINDAN, S., LIU, J., KANSAL, A., AND SIVASUBRAMANIAM, A. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the ACM Symposium on Cloud Computing* (2011).

[13] GREENBERG, A., HAMILTON, J., MALTZ, D. A., AND PATEL, P. The cost of a cloud: research problems in data center networks. *In ACM SIGCOMM Comput. Commun. Rev. 39*, 1 (January 2009), 68–73.

[14] HAMILTON, J. Cost of power in large-scale data centers. Blog entry dated 11/28/2008 at `http://perspectives.mvdirona.com`, 2009. Also in Keynote, at ACM SIGMETRICS 2009.

[15] HASSIN, R., AND LEVIN, A. A better-than-greedy approximation algorithm for the minimum set cover problem. *SIAM Journal on Computing 35*, 1 (2005), 189–200.

[16] IYER, R., ILLIKKAL, R., TICKOO, O., ZHAO, L., APPARAO, P., AND NEWELL, D. Vm3: Measuring, modeling and managing vm shared resources. *Computer Networks 53*, 17 (2009), 2873–2887.

[17] JIANG, Y., SHEN, X., CHEN, J., AND TRIPATHI, R. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the seventeenth International Conference on Parallel Architectures and Compilation Techniques* (2008).

[18] JIANG, Y., TIAN, K., AND SHEN, X. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers* (2010), pp. 6–8.

[19] KOH, Y., KNAUERHASE, R., BRETT, P., BOWMAN, M., WEN, Z., AND PU, C. An analysis of performance interference effects in virtual environments. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2007).

[20] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the thirteenth annual ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2007), pp. 959–967.

[21] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the forty-fourth annual IEEE/ACM International Symposium on Microarchitecture* (2011).

[22] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the USENIX International Conference on Virtual Execution Environments* (2005).

[23] NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the fourth ACM European Conference on Computer Systems* (2010).

[24] RIGHTSCALE. RightScale scalable websites. `http://www.rightscale.com/solutions/cloud-computing-uses/scalable-website.php`.

[25] ROYTMAN, A., KANSAL, A., GOVINDAN, S., LIU, J., AND NATH, S. Algorithm design for performance aware VM consolidation. Tech. rep., Microsoft Research, 2013. Number MSR-TR-2013-28.

[26] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. In *O'REILLY: Web Performance and Operations Conference (Velocity)* (2009).

[27] SOARES, L., TAM, D., AND STUMM, M. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proceedings of the forty-first annual IEEE/ACM International Symposium on Microarchitecture* (2008).

[28] SRIKANTAIAH, S., KANSAL, A., AND ZHAO, F. Energy aware consolidation for cloud computing. In *Proceedings of the workshop on Power Aware Computing and Systems* (2008), HotPower.

[29] TIAN, K., JIANG, Y., AND SHEN, X. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the ACM International Conference on Computing Frontiers* (2009).

[30] TREVISAN, L. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of the thirty-third annual ACM Symposium on Theory of Computing* (2001).

[31] VERMA, A., AHUJA, P., AND NEOGI, A. pmapper: power and migration cost aware application placement in virtualized systems. In *Middleware* (2008).

[32] VERMA, A., AHUJA, P., AND NEOGI, A. Power-aware dynamic placement of hpc applications. In *Proceedings of the International Conference on Supercomputing* (2008).

[33] VMWARE. Vmware distributed power management concepts and use. `http://www.vmware.com/files/pdf/DPM.pdf`.

[34] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. *Trans. Storage 2*, 3 (2006), 283–308.

[35] ZHAO, L., IYER, R., ILLIKKAL, R., MOSES, J., MAKINENI, S., AND NEWELL, D. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Proceedings of the sixteenth International Conference on Parallel Architectures and Compilation Techniques* (2007).

Working Set-based Physical Memory Ballooning

Jui-Hao Chiang
Stony Brook University

Han-Lin Li and Tzi-cker Chiueh
Industrial Technology Research Institute

Abstract

Minimizing the total amount of physical memory consumption of a set of virtual machines (VM) running on a physical machine is the key to improving a hypervisor's *consolidation ratio*, which is defined as the maximum number of VMs that can run on a server without any performance degradation. To give each VM just enough physical memory equal to its true working set (TWS), we propose a *TWS-based memory ballooning* mechanism that takes away all unneeded physical memory from a VM without affecting its performance. Compared with a state-of-the-art commercial hypervisor, this working set-based memory virtualization technique is able to produce noticeably more effective reduction in physical memory consumption under the same input workloads, and thus represent promising additions to the repertoire of hypervisor-level optimization technologies.

1 Introduction

Memory virtualization enables the hypervisor to allocate to each running VM just enough physical memory without performance degradation (memory ballooning) and consolidate physical memory pages with identical contents across VMs (memory deduplication [6, 10, 18, 16]). These optimization techniques make the best of the available physical memory on a virtualized server and maximize the number of VMs that could run on it, or the *consolidation ratio*. Because memory deduplication is an important technique used in both commercial and open-source hypervisors [21, 8] and has been extensively dealt with in a separate paper [13], this paper focuses only on memory ballooning.

When a VM is started, the amount of physical memory that the hypervisor gives to the VM is equivalent to that specified in its configuration file. However, in most cases VMs do not use up all the given memory because VMs tend to be provisioned conservatively. By definition, the amount of physical memory that a VM needs at any point in time is its working set size at that instant. Therefore, if there exists a way to accurately estimate a VM's working set size, the hypervisor could leverage this estimate to take away unneeded memory pages from the VM using the memory ballooning mechanism [21, 8, 20].

This paper describes the design, implementation and evaluation of an intelligent memory ballooning algorithm based on the working set size information of running VMs. To derive the working set size of a given VM, we exploit the page reclamation mechanism built into the guest OS by iteratively decreasing the VM's physical memory allocation until it starts swapping in pages. When we say a VM's current working set size is X, we meant the size of the memory pages the VM is going to access in the next observation window is X. In our design, the observation window is set to 1 second.

2 Working Set Estimation

The physical memory given to a VM on a virtualized server at the start-up time forms the VM's *guest physical address space*, which is mapped to the server's *machine physical address space* through a mapping table, the Extended Page Table (EPT) in the case of the X86 architecture. The working set of a VM is defined as the set of memory pages in the guest physical address space that are being actively used by the VM in the recent past [21]. If a VM's working set is a proper subset of the VM's guest physical address space, some physical memory pages allocated to the VM could be safely reclaimed. Even when a VM's exact working set is not available, being able to estimate the working set's size is still useful.

A naive way to determine a VM's working set is to intercept memory accesses made by the VM, for example, marking a VM's memory pages as not-present in the EPT so as to trap and record the number of accesses to each of its pages. The working set of a VM is the set of memory pages that have been accessed at least once in the observation window. However, this scheme is infeasible because the overhead of trapping every memory read/write is simply too prohibitive to be acceptable in practice. To get around this problem, VMware's ESX used a sampling approach to estimating the working set size of a VM. Periodically it marks a randomly sampled subset of the VM's guest physical pages as invalid, counts the number of pages in the subset that are accessed whenever a protection fault against any of these pages occurs, and uses the resulting count to infer the VM's working set size.

Another way to estimate a VM's working set size, used by the *self-ballooning* mechanism [15] in the Xen hypervisor, is to directly use the `Committed_AS` statistic maintained by the Linux kernel, which corresponds to the total number of *anonymous* memory pages consumed by all processes on a VM. For page reclamation, Linux maintains two LRU (Least Recently Used) lists, *Active* and *Inactive*, for each of the following two types of memory pages: (1) *Anonymous Memory*, which corresponds to the heaps and stacks of user processes, and (2) *Page Cache*, which corresponds to the kernel's memory to buffer and cache the payloads of disk reads and writes.

Utilizing the hardware reference bit, Linux puts pages that are accessed more frequently into Active list and leave pages that are accessed less frequently in Inactive list. The page reclamation mechanism traverses the Inactive list to free its pages and possibly re-allocate them. If a reclaimed page belongs to anonymous memory, the kernel marks the page's page table entry as non-present, and swaps out the page's content to the swap disk. When the page is later accessed, a *swapin* event occurs and it is swapped in. If a reclaimed page belongs to page cache, the kernel flushes its content to disk if it has been dirtied. If the page is later accessed, a *refault* event occurs and it is brought back in.

When a VM's physical memory allocation is larger than or equal to its working set size, the number of swapin and refault events should be close to zero. This observation inspires the third way to estimate a VM's working set size: Gradually decreasing the balloon target of the balloon driver in the VM until the VM's swapin and refault counts start to become non-zero. The amount of physical memory allocated to the VM at that instant is the VM's working set size. More concretely, a 3-state finite state machine, as shown in Figure 1, is used to adaptively track a VM's working set size (WSS). Anytime the WSS changes, we adjust the VM's balloon target accordingly. The finite-state machine starts in the *FAST* state and initializes the VM's WSS to the VM's `Committed_AS`. While in the *FAST* state, the finite-state machine iteratively lowers the VM's WSS by 5% of the

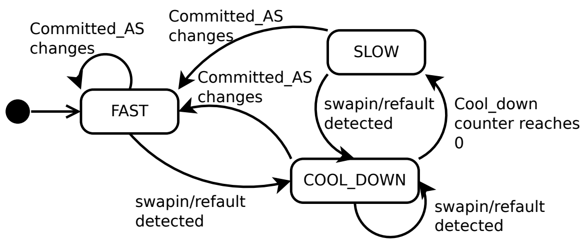


Figure 1: *The finite-state machine used to track a VM's working set size.*

current `Committed_AS` value at the end of every epoch (epoch size set to 1 second currently), until swapin or refault events occur within the current epoch, which suggests the finite-state machine may have overshot the WSS adjustment. As soon as swapin/refault events arise in an epoch, the finite-state machine raises the VM's current WSS estimate by the sum of the observed swapin and refault event counts, and enters the *COOL_DOWN* state, regardless of whether the finite-state machine was originally in the *FAST*, *COOL_DOWN* or *SLOW* state.

While in the *COOL_DOWN* state, the finite-state machine initializes a cool-down counter to a default timeout value (currently set at 8 seconds) and waits for it to expire, and resets the cool-down counter to the same default value if additional swapin/refault events arise. In the *SLOW* state, the finite-state-machine applies the same logic as in *FAST* state except that the VM's WSS is iteratively lowered by 1% of the current `Committed_AS` value in each epoch. Whenever the tracked VM's `Committed_AS` changes, the finite-state machine considers the VM's working set size has changed significantly, and resets itself by entering the *FAST* state and re-initializing the VM's WSS to the new `Committed_AS`.

3 TWS-based Memory Ballooning

Memory ballooning [21, 8] is a technique that reclaims physical memory from a VM by installing inside the VM a balloon driver that allocates memory pages from the VM's kernel via the standard APIs, pins them down, and returns them to the hypervisor. The balloon target of a balloon driver is the difference between the VM's configured memory requirement and the amount of memory it allocates from the VM.

How to correctly set a VM's balloon target is an important issue. When a balloon driver allocates more than the host VM's free memory pool, the VM OS's page reclamation mechanism is triggered to evict cold pages. The upper bound on a VM's balloon target is the VM's configured memory requirement, and the lower bound is the VM's minimum memory requirement that prevents Out-of-Memory exceptions. The optimal way to set a VM's balloon target is to set it to the VM's working set size, because this allows the hypervisor to reclaim the maximum amount of physical memory from a VM while reducing the performance impact on the VM to the minimum.

The self-ballooning mechanism in the Xen hypervisor sets a Linux VM's balloon target to its current `Committed_AS` value. This approach guarantees that applications consuming anonymous memory not suffer from any swap-in delay because all their stacks and heaps are likely to be memory-resident. However, compared with the working set-based approach to setting the balloon target, this approach has two deficiencies. First, `Committed_AS` does not factor the page cache into a VM's physical memory demand, and thus may cause substantial performance degradation for applications with intensive disk I/O activities, which could significantly benefit from the page cache. In contrast, the working set approach keeps a counter for refault events, and incorporates this counter into the calculation of a VM's working set size and thus balloon target. Second, `Committed_AS` captures only the pages that are allocated but not those that are actually used recently. More specifically, `Committed_AS` is incremented upon the first access to each newly allocated anonymous memory page and is decremented only when the owner process explicitly frees the page. For example, if a program allocates and accesses a memory page only once when the program starts but leaves it untouched until the program exits, the Linux kernel cannot exclude this cold page from a VM's `Committed_AS` even though it is clearly outside the VM's working set. In contrast, the working set approach actively forces the VM OS to invoke its page reclamation mechanism to pinpoint and evict cold pages.

4 Performance Evaluation

In this paper, we report the results of a performance evaluation study of TWS-based memory ballooning. The test machine used in this study contains an Intel Core i7 quad-core processor with VT and EPT enabled and 16 GB physical memory, and runs Xen-4.1 with 64-bit vanilla Linux 3.2.6 as the *Dom0* kernel. All the VMs in this study are configured with 1 virtual CPU and 2GB memory, and run Linux 3.2.6 64-bit kernel with the our developed *zballoond* kernel module for memory ballooning. *Zballoond* is a kernel thread that wakes up every second to collect relevant information, such as *Committed_AS*, swapin count and refault count, and make adjustments to the balloon target.

To verify the effectiveness of these TWS-based ballooning algorithm, we first compared it with self-ballooning mechanism in the Xen hypervisor. Then we compared it with the latest VMware ESXi 5.0 server.[1]

Benchmark Used	**TWS Ballooning**		**Self Ballooning**	
	Degradation	**Target**	**Degradation**	**Target**
SPECweb	0%	263.3MB	0%	263.3MB
SPECcpu	3.08%	783.6MB	4.11%	922.6MB
OLTP	3.31%	350.8MB	17.99%	328.8MB

Table 1: *Comparison between TWS-based ballooning and self ballooning in terms of performance degradation and balloon target for the three benchmarks, SPECweb Banking, SPEC CPU 401 and OTLP. The performance degradation is calculated based on a comparison with the performance of the same VM that is configured with 2GB memory.*

In this comparison, we used two identical test machines where one runs the Xen hypervisor with the TWS-based memory virtualization optimizations and the other runs the ESXi server. The memory given to each VM does not include anything owned by the hypervisor.

4.1 Effectiveness of TWS-based Ballooning

We evaluate the effectiveness of TWS-based ballooning by comparing the performance degradation and balloon target of a VM running a set of benchmark programs when TWS-based ballooning is used with those when Xen's self-ballooning is used. The balloon target of a VM is the amount of physical memory that a memory ballooning scheme allocates to the VM. The performance degradation of a memory ballooning scheme is the performance difference between a benchmark program running in a VM whose physical memory allocation is controlled by the ballooning scheme in question and the same benchmark program running in a VM that is configured with and indeed given 2GB memory, or the *Baseline* configuration. The following three benchmark programs are used: *SPECweb Banking* [3] running against Apache [1], *SPEC CPU*, and *OLTP* from the Sysbench suite [4] running against MySQL [2].

Table 1 shows the performance degradation and balloon target comparison between TWS-based ballooning and self-ballooning for the three benchmark programs. The memory requirement of SPECweb Banking benchmark is smaller than the minimum physical memory allocation to the test VM, 263.3MB. As a result, both TWS-based ballooning and self-ballooning produce the same balloon target, which is the same as the minimum physical memory allocation, and the benchmark program does not experience any performance degradation under TWS-based ballooning and under self-ballooning, when compared with the Baseline configuration. For the SPEC CPU 401 benchmark, the average balloon target of TWS-based ballooning is 15.07% (783.6MB vs. 922.6MB)

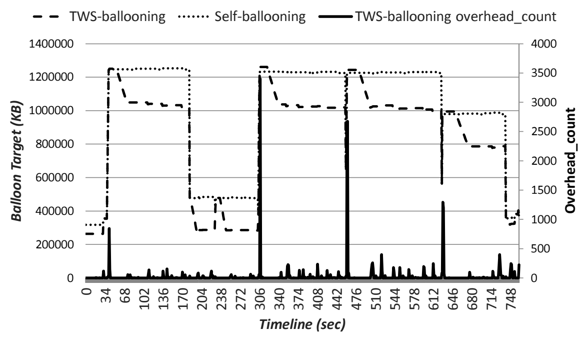


Figure 2: *The balloon targets produced by TWS-based ballooning and self-ballooning over time, and the resulting combined swapin and refault count over time under TWS-based ballooning, when the SPEC CPU 401 benchmark is used as the test workload.*

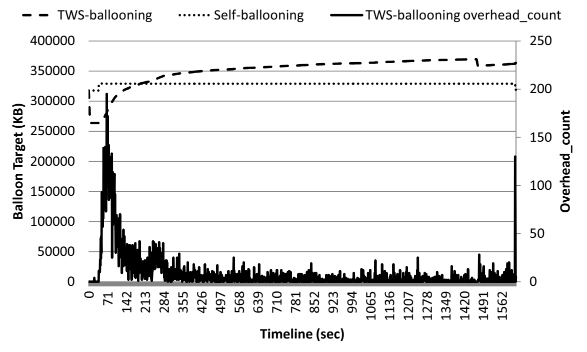


Figure 3: *The balloon targets produced by TWS-based ballooning and self-ballooning over time, and the resulting combined swapin and refault count over time under TWS-based ballooning, when the Sysbench OLTP benchmark is used as the test workload.*

smaller than that of self-ballooning, and yet the performance degradation of TWS-based ballooning is smaller than that of self-ballooning (3.08% vs. 4.11%).

The superiority of TWS-based ballooning comes from the fact that the working set size it produces effectively removes pages that are allocated but unused, as shown by the gap between the two balloon target curves in Figure 2. However, despite allocating a smaller amount of physical memory to the test VM, the performance degradation of TWS-based ballooning is smaller than self-ballooning, because it reacts faster to the sudden change in the VM's demand, e.g. at time points 320 seconds, 460 seconds, and 630 seconds of Figure 2. During these transitions, TWS-based ballooning is able to allocate more physical memory than Committed_AS, and thus cuts down unnecessary swapin and refault events.

Because the OLTP benchmark performs intensive disk I/O accesses and thus requires a larger page cache, Committed_AS is not an accurate estimate of the benchmark's working set as it does not take into account page cache. As a result, the average balloon target produced by TWS-based ballooning is 6.70% higher than self-ballooning, and justifiably so, because the performance degradation of TWS-based ballooning is only 3.31%, which is significantly smaller than that of self-ballooning, or 17.99%. As shown in Figure 3, TWS-based ballooning detects refault events and increases the test VM's balloon target accordingly, and as a result produces a balloon target that is more in line with the VM's working set size and more capable of reducing the performance overhead of memory ballooning to the minimum.

We also run two VMs, one with a constant working set size of 300MB and the other with a constant working set size of 1200MB, on the Xen hypervisor with TWS-based ballooning and on VMware's ESXi 5.0. Each VM is configured with 2 GB memory but given only 263.3MB at the start-up time. After these two VMs start to run, it takes TWS-based ballooning 10 seconds to reach the ideal physical memory allocation, i.e., giving 300MB to the 300MB VM and giving 1200MB to the 1200MB VM. However, for the same set-up, it takes VMware ESXi 136 seconds to reach the same ideal physical memory allocation. The reason that VMware ESXi takes longer to accomplish the same is because it uses a sampling approach to probe a VM's working set size.

5 Related Work

Standard operating systems estimate the active portion of buffer cache or page cache by maintaining LRU-like statistics [19, 12, 5] to implement page replacement logic. Lu et al. [14] proposed to allocate a small portion of memory to each VM while leaving the remaining memory as an exclusive cache is managed by the hypervisor. Thus, the memory accesses of VMs can be intercepted within the exclusive cache, and the LRU miss ratio curve [5] is derived to measure the working set size. Zhao et al. [24, 23] track the memory access of VMs by changing the user/supervisor privilege bit of guest page table entries to supervisor mode so that all memory access of VM will be trapped because the VM runs in user mode. Similarly, the LRU miss ratio curve is also derived for working set size prediction.

To reduce the overhead from trapping memory access, the VMware ESX server [21] uses sampling based mechanism to predict the working set size of VMs. To perform the sampling, the ESX server randomly chooses a few hundreds memory pages periodically, e.g., the default setting is to choose 100 pages per 60-second for each VM. However, this mechanism only gives a rough estimation of the VM working set size, and it can not reflect the working set size exceeding the current allocated memory.

When it comes to reclamation mechanism, the Clock algorithm [9] is commonly used in guest OSs and several research efforts [17, 22, 7, 11] aimed to estimate the working set size by monitoring the changes of access bit on the hardware page table. This approach requires modifications to the guest OS. In contrast, our approach leverages the guest OS's page reclamation mechanism and does not require any guest OS modifications.

6 Conclusion

Making efficient utilization of the physical memory available on a virtualized server is a key technical challenge for modern hypervisors. Possible solutions include *memory de-duplication*, which allows different VMs to share common pages, and *memory ballooning*, which reclaims unused pages from a VM when its physical memory allocation is larger than its working set size. This paper describes and evaluates techniques that exploit the knowledge of each VM's working set to deliver more efficient memory ballooning. More concretely, the specific research contributions of this work are

- A low-overhead active probing mechanism that could accurately sense the working set of each VM and track it dynamically,
- An intelligent memory ballooning algorithm that could detect allocated but unused pages and reclaim them, and

Compared with VMware's ESXi, which is a state-of-the-art hypervisor, the proposed working set estimation scheme is more accurate and more responsive to working set changes, but incurs a slight probing overhead, the proposed memory ballooning algorithm is able to quickly reclaim more memory pages without incurring additional performance penalty.

References

[1] Apache http server project. http://httpd.apache.org/.

[2] Mysql: open source database server. http://www.mysql.com/.

[3] Specweb2009. http://www.spec.org/web2009/.

[4] Sysbench: a system performance benchmark. http://sysbench.sourceforge.net/.

[5] ALMÁSI, G., CAŞCAVAL, C., AND PADUA, D. A. Calculating stack distances efficiently. MSP '02, ACM, pp. 37–43.

[6] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. *Increasing memory density by using KSM.* Linux Symposium, 2009, pp. 19–28.

[7] BANSAL, S., AND MODHA, D. S. Car: Clock with adaptive replacement. FAST '04, USENIX Association, pp. 187–200.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. *Xen and the art of virtualization*, vol. 37. ACM, 2003, pp. 164–177.

[9] CORBATO, F. J. A paging experiment with the multics system. In *In Honor of P.M* (1969), Morse, MIT Press, pp. 217–228.

[10] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. OSDI '08.

[11] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: an effective improvement of the clock replacement. ATEC '05, USENIX Association, pp. 35–35.

[12] JIANG, S., AND ZHANG, X. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. SIGMETRICS '02, ACM, pp. 31–42.

[13] JUI-HAO CHIANG, HAN-LIN LI, T.-C. C. Introspection-based memory de-duplication and migration. VEE '13.

[14] LU, P., AND SHEN, K. Virtual machine memory access tracing with hypervisor exclusive cache. USENIX ATC'07, USENIX Association, pp. 3:1–3:15.

[15] MAGENHEIMER, D. Add self-ballooning to balloon driver. discussion on xen development mailing list and personal communication, april 2008.

[16] MAGENHEIMER, D. *Transcendent Memory on Xen*. XenSummit, February 2009, p. 3.

[17] MAUERER, W. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[18] MURRAY, D. G., H, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. ATEC '09.

[19] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec. 22*, 2 (June 1993), 297–306.

[20] SCHOPP, J. H., FRASER, K., AND SILBERMANN, M. J. Resizing memory with balloons and hotplug. *Linux Symposium 2* (2006), 313319.

[21] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36* (December 2002), 181–194.

[22] ZHANG, I., GARTHWAITE, A., BASKAKOV, Y., AND BARR, K. C. Fast restore of checkpointed memory using working set estimation. *SIGPLAN Not. 46*, 7 (Mar. 2011), 87–98.

[23] ZHAO, W., JIN, X., WANG, Z., WANG, X., LUO, Y., AND LI, X. Low cost working set size tracking. USENIX ATC'11, USENIX Association, pp. 17–17.

[24] ZHAO, W., AND WANG, Z. Dynamic memory balancing for virtual machines. In *VEE '09* (2009).

Notes

[1] VMware ESXi 5.0.0 build-623860.

Coriolis: Scalable VM Clustering in Clouds

Daniel Campello[*], Carlos Crespo[*], Akshat Verma[†], Raju Rangaswami[*], and Praveen Jayachandran[†]
[*]*Florida International University,* [†]*IBM Research - India*

Abstract

The growing popularity of virtualized data centers and clouds has led to virtual machine sprawl, significantly increasing system management costs. We present *Coriolis*, a scalable system that *analyzes* virtual machine images and automatically clusters them based on content and/or semantic similarity. Image similarity analysis can improve in *planning* many management activities (e.g., migration, system administration, VM placement) and reduce their *execution* cost. However, clustering images based on similarity – content or semantic – requires large scale data processing and does not scale well. *Coriolis* uses (i) asymmetric similarity semantics and (ii) a hierarchical clustering approach with a data access requirement that is linear in the number of images. This represents a significant improvement over conventional clustering approaches that incur quadratic complexity and therefore becoming prohibitively expensive in a cloud setting.

1 Introduction

Cloud computing lends a fundamental shift to how businesses view IT, from being capital-intensive to being a commodity that can be acquired on-demand and paid for as per usage. However, the growing popularity of cloud data centers has led to the problem of virtual machine sprawl. Standardization is a key principle that allows cloud providers to provide services on-demand and at a lower cost than what individual IT departments can do. System management costs reduce with standardization of software at all levels: operating systems, middleware, applications, and management tools [1, 13].

We conjecture that classifying (possibly) diverse virtualized servers in a cloud into clusters of similar virtual machines (VMs) can improve the planning of many system management activities. We classify VM similarity into two types – content similarity and semantic similarity. Content similarity refers to data similarity in the raw files that constitute virtual machines. Semantic similarity refers to the similarity in the operating system, middleware, and application software present in two virtual machines. Several management activities can be planned better to reduce their execution cost using analysis of content and/or semantic similarity.

We develop and evaluate *Coriolis*, a framework for clustering images based on any given notion of similarity. Conventional clustering techniques require at least quadratic data access or worse, prohibitive for cloud environments with a large number of VMs. Further, clustering images based on the conventional symmetric notion of similarity leads to a uniform data access pattern; consequently, caching techniques that leverage popularity or locality for optimizing index lookup in deduplication systems [15, 6] are not applicable. *Coriolis* employs a novel tree-based VM clustering algorithm that consumes time that is only linear in the number of images. The algorithm uses an asymmetric notion of similarity to avoid computing all-pairs similarity values and a hierarchical order to introduce popularity in data access.

2 VM Similarity: Types and Applications

The similarity across VMs in enterprise data centers and clouds has been studied extensively in the context of data deduplication [5, 6, 8, 9, 15]. In this section, we discuss both content and semantic similarity and then discuss how such similarity can be utilized for streamlining system management tasks.

2.1 Content Similarity

The classical notion of similarity is that of *content*, whereby a subset of the bytes contained within the images are identical. Identical content can occur either in the form of whole or partial files [11] and techniques to detect similar content have ranged from whole file and fixed size chunking to more sophisticated variable size chunking [8, 15]. Content similarity is useful in minimizing the amount of data that needs to be managed for a task involving a collection of VMs (e.g., VM backup [14] or Virtual Image Library [3]). A recent large-scale study of VM images in a production IaaS cloud investigates such content similarity [7]. This study found that the distribution of *content similarity across images is skewed* and that individual VM images tend to be similar to a small subset of images than to the entire image population leading to clusters of similar images. They also noted that *computing pair-wise similarity is very expensive* and reported results for only 30% of their image collection due to scalability issues.

2.2 Semantic Similarity

Semantic similarity characterizes the similarity of software functionality within images. Examples of semantically similar software include instances of the same application, different versions of the same application, or even different applications that accomplish the same goal (e.g., MySQL and DB2 which both implement database systems and require database expertise to manage). Causes for semantic similarity include standardization of the software stack in modern enterprises and

Use Case	Content	Semantic
Administrator Allocation	×	✓
Troubleshooting	×	✓
VM Placement	✓	✓
Migration	✓	✓

Table 1: Similarity types relevant for each use case

the popularity of specific types of programming models. As identified in previous work, when enterprises are migrated to the cloud, they are adjusted and standardized so that the same set of agents and processes can be used for management services such as backup recovery, security compliance, and patching [13]. Semantic similarity is useful for streamlined system administration, troubleshooting, and management tasks such as grouped scheduling of maintenance and upgrade engineers leading to lower personnel costs. With the growing problem of virtual image sprawl, administrators find it increasingly difficult to keep track of what software is installed on each VM. Automating the detection of VMs with semantically similar software is thus valuable. Unfortunately, the nature of semantic similarity in enterprise and cloud data centers is not well understood.

2.3 Harnessing Image Similarity

We identify four common system management scenarios that can leverage image similarity to reduce data center costs. The most natural use case is allocation of servers to system administrators for routine maintenance. It has been shown that system administrators can be more efficient and manage up to 80% more servers if the servers have a similar software stack [1]. A second use case is troubleshooting system errors during regular updates in data centers. Troubleshooting in data centers is often akin to manual outlier detection where the engineer attempts to identify servers that responded similarly to the update. Once similar servers are identified, the engineer identifies the difference between the failed server and the successful server to fix the issue. Automated clustering of servers based on semantic similarity can aid such identification. Third, placement of VMs to hosts or to management systems often leverage content for efficiency. Images with high semantic similarity are likely to exhibit higher number of duplicate pages in main memory, which can be deduplicated. Similarly, images with higher content similarity can benefit more from deduplication performed at a shared management server (e.g., vSphere [14]).

The final use case is migration of enterprise applications from one data center to another. Migration is performed in batches or waves, where a certain number of images (e.g., 25) are migrated in one weekend [13]. Migrating images with similar content together can reduce migration time using deduplication. Further, images with similar applications can be reconfigured with fewer application experts, reducing migration cost. Identifying image clusters with both high content and semantic similarity and using them to create waves can help reduce both migration time and cost. Table 1 summarizes the type of similarity relevant for all the use cases.

3 Similarity-based VM Clustering

Clustering is a well-studied problem in computer science. While the problem is NP-hard, various heuristics exist with acceptable clustering performance.

3.1 A Representative Clustering Algorithm

k-means is one of the most popular clustering techniques employed in the real world. The algorithm starts with an initial set of k-clusters and refines them iteratively. Even though multiple variants of the algorithm exist, they all apply two canonical operations in each iteration:

- *Assignment Step:* Assign each element to the cluster with the closest mean. *Distance* computation is the core internal operation, performed k times for each element. If there are N elements to cluster, this requires *kN Distance* operations.
- *Update step:* Calculate the new mean for each cluster. The core step is a *Merge* operation which computes the average for 2 elements along each of the D dimensions. In each iteration, across the k clusters, $N-1$ merge operations are performed.

The worst case time for k-means is exponential in N. For arbitrary set of points in $[0,1]^D$, if each point is independently perturbed by a normal distribution with variance σ^2, then the expected running time of k-means algorithm is bounded by $O(N^3 4k^3 4D^8 \log^4(N)/\sigma^6)$ [4]. Even for simple cases, the best known bounds on average running time are at least $O(N^4)$.

3.2 A *Similarity* Function for Images

In spite of its high computational complexity in number of elements, k-means is popular in practice because the time taken for each *Distance* and *Merge* operation is usually very small. Even for problems with 100 dimensions, *Distance* and *Merge* operations require only about 100 addition and division operations. However, these operations are not very well-defined for VM images. We first define a natural definition of these operations and then present the time taken for each operation.

For VM images, it is more natural to define a *similarity* measure than a distance measure. Two images are similar if they contain a large number of identical elements (files or software). Given a pair of images I_i, I_j, similarity between the images can be defined as

$$SIM(I_i, I_j) = \frac{wt(I_i \cap I_j)}{wt(I_i \cup I_j)} \tag{1}$$

Image Size	Similarity	Merge
8.8 GB	45.5 sec	14.7 sec
12.3 GB	75.2 sec	24.1 sec
13.6 GB	98.5 sec	31.2 sec
16.3 GB	142.3 sec	44.2 sec
19.7 GB	172.2 sec	53.5 sec
22.1 GB	232.7 sec	64.9 sec

Table 2: Time for *Similarity* and *Merge* operations. Images and file are stored in a database making use of appropriate indices for these operations.

where $I_i \cup I_j$ is a meta-image that consists of the union of I_i and I_j, $I_i \cap I_j$ is a meta-image that consists of the intersection of I_i and I_j. The weight (wt) function is defined based on the type of similarity that needs to be computed. To estimate content similarity, the wt function is the sum of all files in the image, weighted by the sizes of the files. To estimate semantic similarity, the wt function is the sum of all software deployed in the image weighted by the complexity of the software. Adopting other notions of similarity is straightforward (e.g., a weighted composition of content and semantic similarity). *Distance* can now be calculated simply as $1 - SIM(I_i, I_j)$. The *Merge* operation would create a new image that constitutes the set of all unique elements across the images.

3.3 Scaling Challenge

We measured the running time for a single *Similarity* and a single *Merge* operation on a dual-core 2 GHz Intel Xeon with 4GB memory and images stored on a 5-disk RAID5 SATA array. Table 2 lists run times for real images of different sizes. While the actual times seem small, in aggregate, the costs of these operations present a significant challenge. For example, a data center with 1000 images would have to perform 1000^3 similarity computations (even for the best special cases on average complexity), and would need about 2000 years.

In-memory data structures can reduce the cost of these operations. We conducted experiments by enabling the in-memory feature in MySQL. We observed that the maximum time taken for one similarity computation is 5 seconds (a reduction of $50X$), which though significant only brings down the similarity computation in our previous example to 40 years. Further, this requires the entire index to be memory resident which is not practical. One could envision computing similarity based on only files that are larger than a certain threshold size in each image, but that again would bring down the running time only by a constant factor, while compromising accuracy.

An alternate approach to speed up clustering is to perform approximate clustering based on pair-wise similarity information. The k-medoids clustering algorithm [12] does exactly that by restricting the cluster center in an iteration to one of the existing points (images). Hence, both assignment and update steps in each iteration can leverage pair-wise similarity values that are computed in advance. This simplifying approximation, however, still requires pair-wise similarity computation for all images. Since individual similarity operations are expensive for VM images, this approach becomes un-affordable in practice for moderate to large numbers of VMs as is typical in a cloud, as we shall demonstrate later (§4.3). Anecdotally, in a recent study on VM image similarity, the authors reported pair-wise similarity only for a fraction of their image corpus citing scalability challenges [7]. With 1000 images, this would take 2 years with the file systems on disk and 15 days with an in-memory system. Clearly, there is a need to reduce the number of operations even further. Unfortunately, k-medoids suffers from an additional challenge, that of determining k a priori. The value of k should ideally be the minimum number of clusters required subject to cluster size constraints dictated by the application. However, this information is not always known a priori. In the next section, we discuss an approach that successfully overcomes the core limitations of existing clustering approaches.

4 *Coriolis*

Coriolis uses a novel approach to VM clustering. We discuss this approach and evaluate its scalability relative to the state-of-the-art k-medoids clustering in this section.

4.1 Solution Idea: Asymmetric Clustering

To solve the computational and memory challenge in VM clustering, we draw on a key insight in *Coriolis*. First, we observe that to significantly speed-up the *Distance* and *Merge* operations, caching only a small subset of the image manifest and hash index of image content must be able to satisfy a large fraction of operations. Enabling cache effectiveness requires introducing asymmetry into the clustering algorithm, that is, the algorithm cannot afford to consider all content from all images as equally important. The *Coriolis* clustering approach involves constructing a tree, where each node in the tree is either a cluster of images or a single image, such that each level in the tree from the root node represents a minimum extent of similarity within images in a cluster. The salient aspects of this approach are:

- **Hierarchical multi-level similarity**: Use multiple levels of similarity to quickly find most relevant clusters. By design, restrict comparisons only with clusters that are similar, reducing the total number of *Similarity* operations.
- **Ordered Index Lookup**: Clusters at low similarity levels are more popular than leaf nodes. Images with popular content will require more accesses and can be cached.
- **Online Clustering:** Add a new node to existing clusters. Allows addition/deletion of images with only incremental computation.

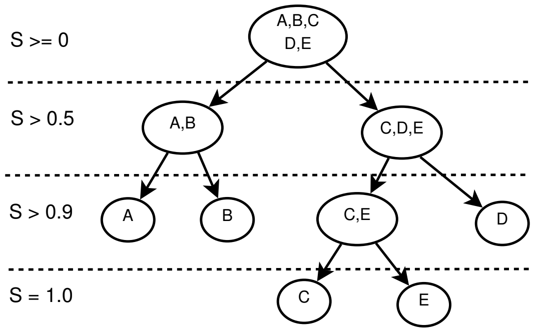


Figure 1: Tree-based clustering. Computed Similarity Values $\{$(A,B):0.75, (C,E):0.95, (CE, D):0.8$\}$

4.2 *Coriolis'* Tree-based Clustering

Coriolis's tree-based clustering approach is outlined below and it is based on two key ideas. The most common operation in clustering is to identify the cluster most similar to a given element and the first idea focuses on speeding up this operation. Since clusters can grow to become very large whereas individual images are typically small, we define and use an asymmetric similarity function S within *Coriolis* that runs in time proportional to the smaller of the two. In particular, we define similarity as the coverage offered by a larger node B (typically a cluster) to a new node A that is being added to the cluster by replacing the *union* operator in the denominator by the *min* operator.

$$S = \frac{wt(A \cap B)}{min(wt(A), wt(B))} \tag{2}$$

Our second key idea is to ensure skew in the usage of images and image clusters allowing effective caching. Further, we reuse the similarity computations done for an image when computing similarity for other images. *Coriolis* uses a tree-based partitioning of the images to achieve both these goals. Each level of the tree represents a predefined minimum level (extent) of similarity. The root of the tree captures a similarity level $S \geq 0$. Thus, all images can be clustered in this meta-node. The last level of the tree captures a similarity level $S = 1$; it consists of either single images or a collection of duplicate images. Intermediate levels represent predefined similarity levels, $0 < S < 1$, which increases with the depth of the tree. We elaborate our representation using the example in Figure 1. Consider 5 images A, B, C, D, E. The tree has 4 levels representing similarity of $0, 0.5, 0.9$ and 1 respectively. A and B have a similarity measure of 0.75. Hence, they are clustered at level $S > 0.5$ but are independent nodes at level $S > 0.9$. Similarly, C and E have a similarity of 0.95 and are grouped together up to all levels $S > 0.9$ but are independent nodes at level $S = 1$.

Given a new image v_i, our goal is to find similar nodes (or meta-nodes) with as few *Similarity* operations as possible. *Coriolis*'s grouping of VM image clusters within a hierarchical tree structure allows early pruning of images that are not similar to the new image v_i. Adding a new image to the *Coriolis* VM image tree, the new image is first added to the root meta-node. Once an image is added to a node, we compute the similarity of the new node with each of its children to determine if it can be added to any child. If the similarity S level is found adequate with more than one child, the new image is only added to the child node with which the similarity is the greatest. If no such child node exists, we create a new child node and add v_i to the node. This process terminates when we reach a leaf node.

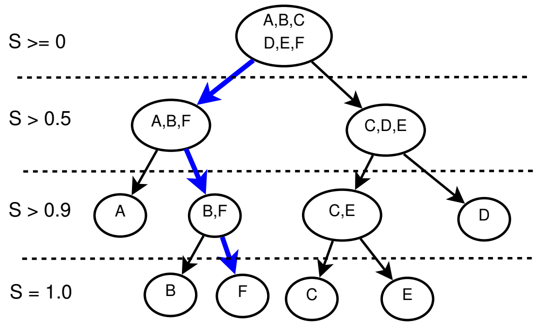


Figure 2: Clustering a new image F. Computed Similarity Values are $\{$(AB,F):0.95, (CDE,F):0.3, (A,F):0.75, (B:F):0.95$\}$

Figure 2 illustrates a new image F as it traverses the tree. It is important to note here that the number of *Similarity* and *Merge* operations executed for an image is proportional to the depth of the tree. The depth of the tree is a pre-defined constant, bound by the *log* of the number of images inserted. Hence, the approach allows us to create a tree in time no more than $O(N \log N)$, where N is the number of images. And given the similarity levels at various tree depths, the tree can then be queried in linear time for clusters with specific properties.

4.3 Scalability Evaluation

To evaluate *Coriolis*, we used VM images from 2 production data centers. The first set of 9 images is from a large-scale enterprise data center at IBM. The latter set of 12 images is from the CS department's small-scale data center at Florida International University. The former set of images are diverse compared to the latter set reflecting the needs typical of a large-scale enterprise data center. Next, we created increasingly larger sets of images from these initial set of 21 production images. We did this by separating out 3 of the 21 images and randomly sampling files contained within these to generate synthetic images. The net effect is that the synthetic images contain a random combination of files from these 3 source images. We performed clustering experiments in a Linux VM configured with 16 GB RAM on an 6-core AMD Opteron processor virtualized using the VMware ESX hypervisor.

We choose k-medoids for this comparison as it is sig-

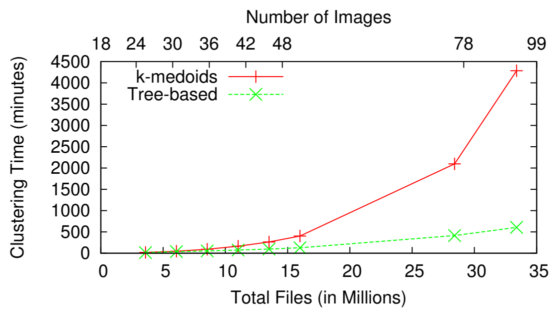


Figure 3: Scalability of k-medoids and tree-based clustering algorithms.

nificantly faster than *k*-means. Fig 3 presents the time taken by the k-medoids algorithm and the tree-based clustering algorithm as the problem size is increased. The time includes the time taken to read file metadata and store it in a database, where similarity and merge operations are performed. The *k*-medoids algorithm takes significantly longer and displays a quadratic increase in clustering time as the number of images is increased. We observed that more than 95% of the time is spent in computing similarity as the cluster size is increased. For clustering 99 images, it takes nearly 3 days, which is clearly unacceptable. In contrast, our tree-based clustering algorithm reduces the number of similarity computations by a factor of 8 and is able to cluster the images within 10 hours; an acceptable window of time even for heavy-weight management VM tasks carried out over weekends.

5 Related Work

Redundancy elimination based on identifying duplicate data is a popular topic of research [10, 15]. Finding similar clusters is a related problem but is more data intensive because it requires processing over the entire index of the data as well as a manifest linking images to their contents. Further, the data access for this problem does not have inherent data popularity and locality, which is used extensively by deduplication techniques for scaling.

The research work closest to ours is VMFlocks which applies standard de-duplication techniques for images that are migrated together across data centers [2]. Given a batch of images, It eliminates raw data duplicates across the given set of VM images. However, it does not tackle identifying images with high redundancy or leveraging semantic similarity.

6 Conclusions

We described the *Coriolis* framework and system that was specifically designed for scalable clustering of VM images so as to counter the negative effects of VM sprawl in cloud data centers. We argued that the state-of-the-art k-medoids clustering algorithm incurs quadratic complexity which we demonstrated as infeasible for cloud scale data centers. *Coriolis*'s distinguishing strength lies in its scalable tree-based image clustering technique that supports an arbitrary similarity metric. This novel technique allows clustering to be performed in $O(N \log N)$ time for a data center with N images, allowing it to scale to large data centers. Our future work will explore the utility of *Coriolis* for data center administrator allocation, troubleshooting, and large-scale VM migration.

Acknowledgments

We thank the paper's reviewers and shepherd, Mustafa Uysal, for helpful feedback. This work was supported by NSF grant CNS-0747038 and an IBM faculty award.

References

[1] IDC Linux Standardization White Paper: Executive Summary. http://www.redhat.com/f/pdf/IDC_Standardize_RHEL_1118_Exec_summary.pdf, 2011.

[2] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: Virtual Machine Co-migration for the Cloud. In *Proc. of the IEEE/ACM HPDC*, June 2011.

[3] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang. Virtual machine images as structured data: the Mirage image library. In *Proc. HotCloud*, 2011.

[4] D. Arthur, B. Manthey, and H. Roeglin. k-means has polynomial smoothed complexity. In *IEEE FOCS*, 2009.

[5] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proc. of the USENIX ATC*, June 2009.

[6] B. Debnath, S. Sengupta, and J. Li. ChunkStash: speeding up inline storage deduplication using flash memory. In *Usenix ATC*, 2010.

[7] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An empirical analysis of similarity in virtual machine images. In *ACM Middleware*, 2011.

[8] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of SysStor*, 2009.

[9] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/ O Performance. In *Proc. of USENIX FAST*, 2010.

[10] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *USENIX FAST*, 2009.

[11] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *Proc. of USENIX FAST*, February 2011.

[12] L. Rousseeuw and L. Kaufman. Clustering by means of medoids. *Statistical data analysis based on the L1-norm and related methods*, 405, 1987.

[13] B. Viswanathan, A. Verma, B. Krishnamurthy, P. Jayachandran, K. Bhattacharya, and R. Ananthanarayanan. Rapid adjustment and adoption to MIaaS clouds. In *ACM Middleware, Industry track*, 2012.

[14] VMWare. VMWare vSphere Data Protection. Technical White Paper, June 2012.

[15] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *USENIX FAST*, 2008.

iShuffle: Improving Hadoop Performance with Shuffle-on-Write

Yanfei Guo, Jia Rao, and Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, USA
$\{$*yguo,jrao,xzhou*$\}$*@uccs.edu*

Abstract

Hadoop is a popular implementation of the MapReduce framework for running data-intensive jobs on clusters of commodity servers. Although Hadoop automatically parallelizes job execution with concurrent map and reduce tasks, we find that, *shuffle*, the all-to-all input data fetching phase in a reduce task can significantly affect job performance. We attribute the delay in job completion to the coupling of the shuffle phase and reduce tasks, which leaves the potential parallelism between multiple waves of map and reduce unexploited, fails to address data distribution skew among reduce tasks, and makes task scheduling inefficient. In this work, we propose to decouple shuffle from reduce tasks and convert it into a platform service provided by Hadoop. We present *iShuffle*, a user-transparent shuffle service that pro-actively pushes map output data to nodes via a novel *shuffle-on-write* operation and flexibly schedules reduce tasks considering workload balance. Experimental results with representative workloads show that iShuffle reduces job completion time by as much as 30.2%.

1 Introduction

Hadoop is a popular open-source implementation of the MapReduce programming model for processing large volumes of data in parallel [7]. Each job in Hadoop consists of two dependent phases, each of which contains multiple user-defined *map* or *reduce* tasks. These tasks are distributed independently onto multiple nodes for parallel execution. The decentralized execution model is essential to Hadoop's scalability to a large number of nodes as map computations can be placed near their input data stored on individual nodes and there is no communication between map tasks.

There are many existing studies focusing on improving the performance of map tasks. Because data locality is critical to map performance, work has been done to preserve locality via map scheduling [21] or input replication[4]. Others also designed interference [5] and topology [14] aware scheduling algorithms for map tasks. While there is extensive work exploiting the parallelism and improving the efficiency in map tasks, only a few studies have been devoted to expedite reduce tasks.

The all-to-all input data fetching phase in a reduce task, known as *shuffle*, involves intensive communications between nodes and can significantly delay job completion. Because the shuffle phase usually needs to copy intermediate output generated by almost all map tasks, techniques developed for improving map data locality are not applicable to reduce tasks [16, 21]. Hadoop strives to hide the latency incurred by the shuffle phase by starting reduce tasks as soon as map output files are available. There is existing work that tries to overlap shuffle with map by proactively sending map output [6] or fetching map output in a globally sorted order [19].

Unfortunately, the coupling of *shuffle* and *reduce* phases in a reduce task presents challenges to attaining high performance in Hadoop clusters and makes existing approaches [6, 19] less effective in production systems. First, in production systems with limited number of *reduce slots*, a job often executes multiple waves of reduce tasks. Because the shuffle phase starts when the corresponding reduce task is scheduled to run, only the first wave of reduce can be overlapped with map, leaving the potential parallelism unexploited. Second, tasks scheduling in Hadoop is oblivious of the data distribution skew among reduce tasks [8, 11, 12], machines running shuffle-heavy reduce tasks become bottlenecks. Finally, in a multi-user environment, one user's long-running shuffle may occupy the reduce slots that would otherwise be used more efficiently by other users, lowering the utilization and throughput of the cluster.

In this paper, we propose to decouple the *shuffle* phase from reduce tasks and convert it into a platform service provided by Hadoop. We present *iShuffle*, a user-transparent shuffle service that overlaps the data shuf-

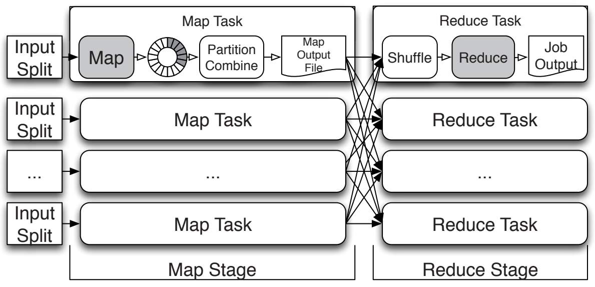


Figure 1: An overview of data processing in Hadoop MapReduce framework.

fling of any reduce task with the map phase, addresses the input data skew in reduce tasks, and enables efficient reduce scheduling. iShuffle features a number of key designs: (1) proactive and deterministic pushing shuffled data from map to Hadoop nodes when map output files are materialized to local file systems, a.k.a, *shuffle-on-write*. (2) automatic predicting reduce execution time based on the input partition size and placing the shuffled data to mitigate the *partition skew* and to avoid hotspots. (3) binding reduce tasks with data partitions only when reduce is scheduled to realize the load balancing enabled by the partition placement.

We implemented iShuffle on a 32-node Hadoop cluster and evaluated its benefits using the Purdue MapReduce Benchmark Suite (PUMA) [2] with datasets collected from real applications. We compared the performance of iShuffle running both shuffle-heavy and shuffle-light workloads with that of the stock Hadoop and a recently proposed approach (i.e., Hadoop-A in [19]). Experimental results show that iShuffle reduces job completion time by 30% and 22% compared with stock Hadoop and Hadoop-A, respectively. iShuffle also achieves significant performance gain in a multi-user environment with heterogeneous workloads.

The rest of this paper is organized as follows. Section 2 introduces the background of Hadoop, discusses existing issues, and presents a motivating example. Section 3 elaborates iShuffle's key designs. Section 4 gives the testbed setup, experimental results and analysis. Related work is presented in Section 5. We conclude this paper in Section 6.

2 Background and Motivation

2.1 Hadoop MapReduce Framework

The data processing in MapReduce [7] model is expressed as two functions: *map* and *reduce*. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key are grouped together and then passed to the same reduce function via *shuffle*, an all-map-to-all-reduce communication. The reduce function processes the intermediate key with the list of its values and generate the final results.

Hadoop's implementation of the MapReduce programming model pipelines the data processing and provides fault tolerance. Figure 1 shows an overview of job execution in Hadoop. The Hadoop runtime partitions the input data and distributes map tasks onto individual cluster nodes for parallel execution. Each map task processes a logical split of the input data that resides on the Hadoop Distributed File System (HDFS) and applies the user-defined map function on each input record. The map outputs are partitioned according to the number of reduce tasks and combined into keys with associated lists of values. A map task temporarily stores its output in a circular buffer and writes the output files to local disk every time the buffer becomes full (i.e., buffer *spill*).

A reduce task consists of two phases: *shuffle* and *reduce*. The shuffle phase fetches the map outputs associated with a reduce task from multiple nodes and merges them into one reduce input. An external merge sort algorithm is used when the intermediate data is too large to fit in memory. Finally, a reduce task applies the user-defined reduce function on the reduce input and writes the final result to HDFS. The reduce phase can not start until all the map phases have finished as the reduce function depends on the output generated by all the map tasks. To overlap the execution of map and reduce, Hadoop allows an early start of the shuffle phase (by scheduling the corresponding reduce task) as soon as 5% of the map tasks have finished.

In the next, we discuss several issues related to shuffle and reduce in the existing Hadoop framework, and give a motivating example showing how these issues affect the performance and efficiency of a Hadoop cluster.

2.2 Input Data Skew among Reduce Tasks

The output of a map task is a collection of intermediate keys and their associated value lists. Hadoop organizes each output file into partitions, one per reduce task and each containing a different subset of the intermediate key space. By default, Hadoop determines which partition a key/value pair will go to by computing a hash value. Since the intermediate output of the same key are always assigned to the same partition, skew in the input data set will result in disparity in the partition sizes. Such a *partitioning skew* is observed in many applications running in Hadoop [8, 11, 12]. Some user-defined partitioner may mitigate the skew but does not guarantee an even data distribution among reduce tasks. As a result, some reduce tasks take significant longer time to complete, slow-

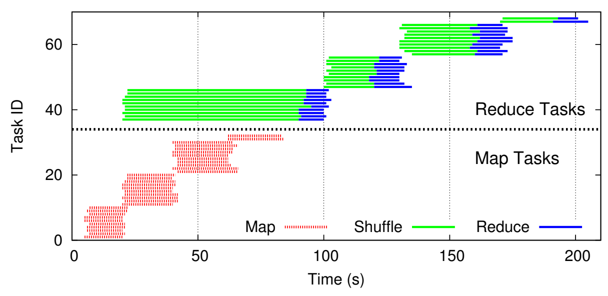


Figure 2: *tera-sort* job execution.

ing down the entire job.

2.3 Inflexible Scheduling of Reduce Tasks

Reduce tasks are created and assigned a task ID by Hadoop during the initialization of a job. The task ID is then used to identify the associated partition in each map output file. For example, shuffle fetches the partition that matches the reduce ID from all map tasks. When there are reduce slots available, reduce tasks are scheduled in the ascending order of their task IDs. Although such a design simplifies task management, it may lead to long job completion time and low cluster throughput. Due to the strict scheduling order, it is difficult to prioritize reduce tasks that are predicted to run longer than others. Further, partitions required by a reduce task may not be generated at the time it is scheduled, occupying the reduce slot and wasting cluster cycles which would otherwise be used by another reduce with all partitions ready.

2.4 Tight Coupling of Shuffle and Reduce

As part of a reduce task, shuffle can not start until the corresponding reduce is scheduled. Besides the inefficiency of job execution, the coupling of shuffle and reduce also leaves the potential parallelism between within and between jobs unexploited. In a production environment, a MapReduce cluster is shared by many users and multiple jobs [21]. Each job only gets a portion of the execution slots and often requires multiple execution waves, each of which consists of one round of map or reduce tasks. Because of the coupling, data shuffling in later reduce waves can not be overlapped with map waves.

Figure 2 shows the execution of one *tera-sort* job with 4GB dataset in a 10-node Hadoop cluster. Each node was configured with 1 map slot and 1 reduce slot. The job was divided into 32 map tasks and 32 reduce tasks [7, 17], resulting in 4 map and reduce waves. We use the duration of the shuffle phase between last execution wave and next reduce phase, termed as *shuffle delay*, to quantify how data shuffling affects the completion of reduce tasks. Due to the overlapped execution, the first reduce wave experienced a shuffle delay of 11 seconds. Unfortunately, remaining reduce waves had on average a delay of 23 seconds before the reduce phase could start. Given that the average length of the reduce phase was 25 seconds, the reduce waves would have been completed in less than half the time if the shuffle delay can be completely overlapped with map.

Figure 2 also suggests that although the overlapping of reduce and map reduced the shuffle delay from 23 to 11 seconds, the first reduce wave occupied the slots three times longer than the following waves. Most time was spent in the shuffle phase waiting for the completion of map tasks. In production systems, allowing other jobs to use these slots may outweigh the benefits brought by the overlapped execution.

These observations revealed the negative impacts of coupling shuffle and reduce on job execution and motivated us to explore a new shuffling design for Hadoop. We found that decoupling shuffle from reduce provides a number of benefits. It enables skew-aware placement of shuffled data, flexible scheduling of reduce tasks, and complete overlapping the shuffle phase with map tasks. In Section 3, we present *iShuffle*, a decoupled shuffle service for Hadoop.

3 iShuffle Design

We propose *iShuffle*, a job-independent shuffle service that pushes the map output to its designated reduce node. It decouples shuffle and reduce, and allows shuffle to be performed independently from reduce. It predicts the map output partition sizes and automatically balances the placement of map output partitions across nodes. iShuffle binds reduce IDs with partition IDs lazily at the time reduce tasks are scheduled, allowing flexible scheduling of reduce tasks.

3.1 Overview

Figure 3 shows the architecture of iShuffle. iShuffle consists of three components: *shuffler*, *shuffle manager*, and *task scheduler*. The shuffler is a background thread that collects intermediate data generated by map tasks and predicts the size of individual partitions to guide the partition placement. The shuffle manager analyses the partition sizes reported by all shufflers and decides the destination of each partition. The shuffle manager and shufflers are organized in a layered structure which is similar to Hadoop's `JobTracker` and `TaskTrackers`. The task scheduler extends existing Hadoop schedulers to support flexible scheduling of reduce tasks. We briefly describe some major features of iShuffle.

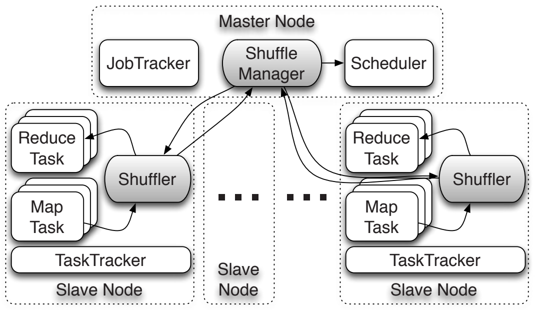


Figure 3: The architecture of iShuffle.

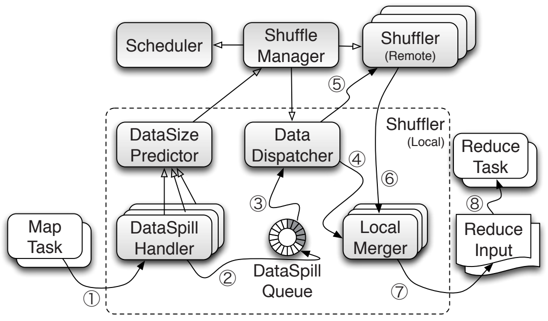


Figure 4: Workflow of Shuffle-on-Write.

User-Transparent Shuffle Service - A major requirement of iShuffle design is the compatibility to existing Hadoop jobs. To this end, we design shufflers and the shuffle manager as job-independent components, which are responsible for collecting and distributing map output data. This design allows the cluster administrator to enable or disable iShuffle through the options in the configuration files. Any user job can use iShuffle service without modifications.

Shuffle-on-Write - The shuffler implements a shuffle-on-write operation that proactively pushes the map output data to different nodes for future reduce tasks every time such data is written to local disks. The shuffling of all map output data can be performed before the execution of reduce tasks.

Automated Map Output Placement - The shuffle manager maintains a global view of partition sizes across all slave nodes. An automated partition placement algorithm is used to determine the destination for each map output partition. The objective is to balance the global data distribution and mitigate the non-uniformity reduce execution time.

Flexible Scheduling of Reduce Tasks - The task scheduler in iShuffle assigns a partition of a reduce task only when the task is dispatched to a node with available slots. To minimize reduce execution time, iShuffle always associates partitions that are already resident on the reduce node to the scheduled reduce.

3.2 Shuffle-on-Write

iShuffle decouples *shuffle* from a *reduce* task and implements data shuffling as a platform service. This allows the shuffle phase to be performed independently from map and reduce tasks. The introduction of iShuffle to the Hadoop environment presents two challenges: user transparency and fault tolerance.

Besides user-defined *map* and *reduce* functions, Hadoop allows customized *partitioner* and *combiner*. To ensure that iShuffle is user-transparent and does not require any change to the existing MapReduce jobs, we design the *Shuffler* as an independent component in the `TaskTracker`. It takes input from the combiner, the last user-defined component in map tasks, performs data shuffling and provides input data for reduce tasks. The shuffler performs data shuffling every time the output data is written to local disks by map tasks, thus we name the operation *shuffle-on-write*.

Figure 4 shows the workflow of the Shuffler. It has three stages: (1) map output collection (step ①②); (2) data shuffling (step ③④⑤⑥); (3) map output merging (step ⑦⑧).

Map output collection - The shuffler contains multiple `DataSpillHandler`, one per map task, to collect map output that has been written to local disks. Map tasks write the stored partitions to the local file system when a spill of the in-memory buffer occurs. We intercept the writer class `IFile.Writer` in the combiner and add a `DataSpillHandler` class to it. While the default writer writing a spill to local disk, the `DataSpillHandler` copies the spill to a circular buffer, `DataSpillQueue`, from where data is shuffled/dispatched to different nodes in Hadoop. During output collection, the `DataSizePredictor` monitors input data sizes and resulted partition sizes, and reports these statistics to the shuffle manager.

Data shuffling - The shuffler proactively pushes data partitions to nodes where reduce tasks will be launched. Specifically, a `DataDispatcher` reads a partition from the `DataSpillQueue` and queries the shuffle manager for its destination. Based on the placement decision, a partition could be dispatched to the shuffler on a different node or to the `local merger` in the same shuffler.

Map output merging - The map output data shuffled at different times needs to be merged to a single reduce input file and sorted by key before a reduce task can use it. The `local merger` receives remotely and locally shuffled data and merges the partitions belonging to the same reduce task into one reduce input. To ensure correctness, the `merger` only merges partitions from successfully finished map tasks.

3.2.1 Fault Tolerance

iShuffle is robust to the failure of map and reduce tasks. Similar to [6], iShuffle maintains a bookkeeping of spill files from all map tasks. If a map task fails, its data spills in the `DataSpillQueue` and `merger` will be discarded. The `merger` merges partitions only when the corresponding map tasks commit their execution to the JobTracker. This prevents reduce tasks from using incomplete data. We also keep the merged reduce inputs in the `merger` until reduce tasks finish. In case of a failed reduce task, a new reduce can be started locally without fetching all the needed map output.

3.3 Automated Map Output Placement

The shuffle-on-write workflow relies on key information about the partition placement for each running job. The objective of partition placement is to balance the distribution of map output data across different nodes, so that the reduce workloads on different nodes are even. The optimal partition placement can be determined when the sizes of all partitions are known. However, this requires that all map tasks are finished when making the placement decisions, which effectively enforce a serialization between map tasks and the shuffle phase. iShuffle estimates the final partition sizes based on the amount of processed input data and current partition size, and uses the estimation to guide partition placement.

3.3.1 Prediction of Partition Sizes

The size of a map output partition depends on the size of its input dataset, the map function, and the partitioner. Verma *et al.* [18], found that the ratio of map output size and input size, also known as *map selectivity*, is invariant given the same job configuration. As such, the partition size can be determined using the metric of map selectivity and input data size. The shuffle manager monitors the execution of individual map tasks and estimates the map selectivity of a job by building a mathematical model between input and output sizes.

For a given job, the input dataset is divided into a number of logical splits, one per map task. Since individual map tasks run the same map function, each map task shares the same map selectivity with the overall job execution. By observing the execution of map tasks, where a number of input/output size pairs are collected, shuffle manager builds a model estimating the map selectivity metric. Shuffle manager makes k observations of the size of each map output partition. As suggested in [18], it derives a linear model between partition size and input data size:

$$p_{i,j} = a_j + b_j \cdot D_i, \tag{1}$$

where $p_{i,j}$ is the jth partition size in the ith observation and D_i is the corresponding input size. We use linear regression to obtain the parameters for m partitions, one per reduce task. Since MapReduce jobs contain many more map tasks than reduce tasks (as shown in Table 1), we are able to collect sufficient samples for building the model. Once a model is obtained, the final size of a map output partition can be calculated by replacing D_i with the actual input size of the map task.

3.3.2 Partition Placement

With predicted partition sizes, the shuffle manager determines the optimal partition placement that balances reduce workload on different nodes. Because the execution time of a reduce task is linear to its input size, evenly placing the partitions leads to balanced workload. Formally, the partition placement problem can be formulated as: given m map output partitions with sizes of $p_1, p_2, \ldots, p_m$, find the placement on n nodes, $S_1, S_2, \ldots, S_n$, that minimizes the placement difference:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\mu - \sum_{j\in S_i} p_j\right)}, \tag{2}$$

where μ is the average data size on one node.

```
Data: $p$: list of partition
Data: $S$: list of nodes, has the size of all allocated
      partitions
Result: Balanced partition placement
sort list $p$ in descending order of partition sizes;
for $i \leftarrow 1$ to $m$ do
    $min\_node \leftarrow S[1]$;
    for $j \leftarrow 1$ to $n$ do
        if $S[j].size < min\_node.size$ then
            $min\_node \leftarrow S[j]$;
        end
    end
    $min\_node$.place($p[i]$);
end
```

Algorithm 1: Partition placement.

Partition placement problem can be viewed as the load balancing problem in multiprocessor systems [9] and is thus NP-hard. While the optimal solution can be prohibitively expensive to attain, we propose a heuristic-based approach to approximate an optimal placement. The detail of this approach is presented in Algorithm 1. This algorithm is based on two heuristics, the `largest partition first` for picking partitions and the `less`

Table 1: Benchmark details.

Benchmark	Input Size (GB)	Input Data	# of Maps	# of Reduce	Shuffle Volume (GB)
self-join	250	synthetic	4000	180	246
tera-sort	300	synthetic, random	4800	180	300
ranked-inverted-index	220	multi-word-count output	3520	180	235
k-means	30	Netflix data, $k = 6$	480	6	32
inverted-index	250	Wikipedia	4000	180	57
term-vector	250	Wikipedia	4000	180	59
wordcount	250	Wikipedia	4000	180	49
histogram-movies	200	Netflix data	3200	180	0.002
histogram-ratings	200	Netflix data	3200	180	0.0012
grep	250	Wikipedia	4000	180	0.0013

`workload first` for picking destination nodes. It sorts the partitions in the descending order of size and assigns the largest partition to the nodes with the least aggregate data size. It repeats until all the partitions are assigned.

3.4 Flexible Reduce Scheduling

In Hadoop, reduce tasks are assigned map output partitions statically during job initialization. When there are reduce slots available on idle nodes, reduce tasks are dispatched according to the ascending order of their task IDs. This restriction on reduce scheduling leads to inefficient execution where reduces that are waiting for map tasks to finish occupy the slots for a long time. Because iShuffle proactively pushes output partitions to nodes, it requires that reduce tasks are launched on nodes that hold the corresponding shuffled partitions. To this end, iShuffle breaks the binding of reduce tasks and map output partitions and provides flexible reduce scheduling.

An intuitive approach for flexible reduce scheduling is to traverse the task queue and find a reduce that has shuffled data on the requesting node. However, this approach does not guarantee that there is always a “local” reduce available for dispatching. iShuffle employs a different approach that assigns partitions to reduce tasks at the time of dispatching. For single-user clusters, we modified Hadoop’s FIFO scheduler to support the runtime task-partition binding. When a node with available reduce slots requests for new reduce tasks, the `task scheduler` first check with the shuffle manager to obtain the list of partitions that reside on this node. The scheduler picks the first partition in the list and associates its ID with the first reduce task in the waiting queue. The selected reduce task is then launched on the node. As such, all reduce tasks are guaranteed to have local access to their input data.

For multi-user clusters with heterogeneous workloads, we add the support for runtime task-partition association to the Hadoop Fair Scheduler (HFS). The minimum fair share allocated to individual users can negatively affect the efficiency of iShuffle as reduce tasks may be launched on remote nodes to enforce fairness. We disable such fairness enforcement for reduce tasks to support more flexible scheduling. This allows some users to temporarily run more reduce tasks than others. We rely on the following designs to preserve fairness among users and avoid starvation. First, the fair share of map tasks is still in effect, guaranteeing fair chances for users to generate map output partitions. Second, while records are sorted by key within each partition after shuffling, partitions belonging to different users are randomly placed in the list, giving each user an equal opportunity to launch reduce tasks. Finally and most importantly, reduce tasks are started only when all their input data is available. This may temporarily violates fairness, but prevents wasted cluster cycles spent in waiting for unfinished maps and results in more efficient job execution.

4 Evaluation

4.1 Testbed Setup

Our testbed was a 32-node Hadoop cluster. Each node had one 2.4 GHz 4-core Intel Xeon E5530 processor and 4 GB memory. All nodes were interconnected by a Gigabit Ethernet. The operating system uses Linux kernel 2.6.24. We deployed Hadoop stable release version 1.1.1 and each machine ran Ubuntu Linux with kernel 2.6.24. Two nodes were configured as the `JobTracker` and `NameNode`, respectively. The rest 30 nodes were configured as slave nodes for HDFS storage and MapReduce task execution. We set the HDFS block size to its default value 64 MB. Each slave node was configured with 4 map slots and 2 reduce slots, resulting in a total capacity of running 120 map and 60 reduce tasks simultaneously in the cluster.

For comparison, we also implemented Hadoop-A proposed in [19]. It enables reduce tasks to access map output files on remote disks through the network. By using a priority queue-based merge sort algorithm, Hadoop-A eliminates repetitive merge and disk accesses, and removes the serialization between the shuffle and reduce phases. However, Hadoop-A requires the remote direct memory access (RDMA) feature on Infiniband interconnections for fast remote disk access. We implemented Hadoop-A using remote procedure calls on our testbed with Gigabit Ethernet and compared its performance with iShuffle on commodity hardware.

4.2 Workloads

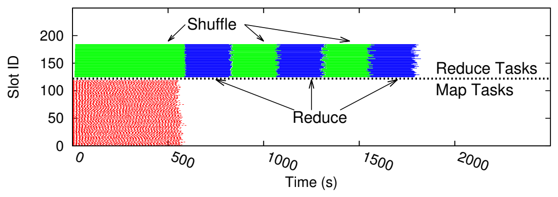
We used the Purdue MapReduce Benchmark Suite (PUMA) [2] to compose workloads for evaluation. PUMA contains various MapReduce benchmarks and real-world test inputs. Table 1 shows the benchmarks and their configurations used in our experiments. For most of the benchmarks, the number of reduce tasks was set to 180 to allow multiple reduce waves. The only exception was *k-means*, which ran on a 30 GB dataset with 6 reduce tasks.

These benchmarks can be divided into two categories: shuffle-heavy and shuffle-light. Shuffle-heavy benchmarks have high map selectivity and generate a large volume of data to be exchanged between map and reduce. Thus, such benchmarks are sensitive to optimizations on the shuffle phase. For shuffle-light benchmarks, there is little data that needs to be shuffled. We used both benchmark types to evaluate the effectiveness of iShuffle and its overhead on workloads with little communications.

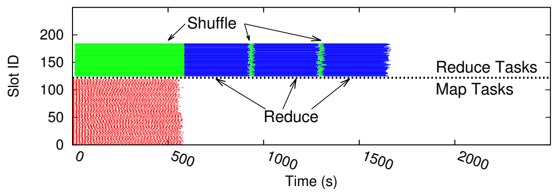
4.3 Reducing Shuffle Delay

Recall that we defined shuffle delay as the duration between the last wave of execution and the next reduce wave. Shuffle delay measures the shuffle period that can not be overlapped with the previous wave. The smaller the shuffle delay, the more efficient the shuffling scheme. We ran *tera-sort* on stock Hadoop, Hadoop-A and iShuffle, and recorded the start and completion times of each map, shuffle and reduce phase.

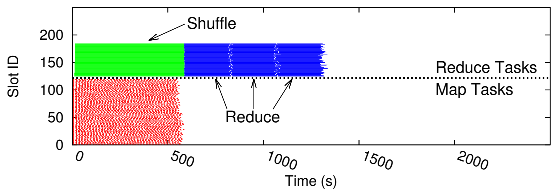
Figure 5 shows the trace of the *tera-sort* job execution under different approaches. The X-axis is the time span of job execution and Y-axis represents the map and reduce slots. The results show that iShuffle had the best performance with 30.2% and 21.9% shorter job execution time than stock Hadoop and Hadoop-A, respectively. As shown in Figure 5(a), there is a significant delay of the reduce phase for every reduce task in stock Hadoop. Due



(a) Hadoop.



(b) Hadoop-A.



(c) iShuffle.

Figure 5: Execution trace of *tera-sort* using stock Hadoop, Hadoop-A, and iShuffle approaches.

to proactive placement of map output partitions, iShuffle had almost no shuffle delays. Note that Hadoop-A also significantly reduced shuffle delay because it operates on globally sorted partitions and can greatly overlap the shuffle and reduce phase.

iShuffle outperformed Hadoop-A on our testbed for two reasons. First, the building of the priority queue poses extra delay, e.g., the shuffle delay before the second and third reduce waves in Hadoop-A, to each reduce task. Second, the remote disk access in an Ethernet environment is significant slower than that in an Infiniband network, which leads to much longer reduce phases in Hadoop-A.

4.4 Reducing Job Completion Time

We study the effectiveness of iShuffle in reducing overall job completion time with more comprehensive benchmarks. We use the job completion time in stock Hadoop implementation as the baseline and compare the normalized performance of iShuffle and Hadoop-A. Figure 6 shows the normalized job completion time of all benchmarks listed in Table 1. The results show that for shuffle-heavy benchmarks such as *self-join*, *tera-sort*, and *ranked-inverted-index*, iShuffle outperformed the stock Hadoop by 29.1%, 30.1%, and 27.5%, respec-

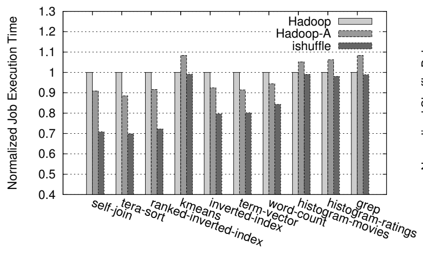


Figure 6: Job completion time using three different approaches.

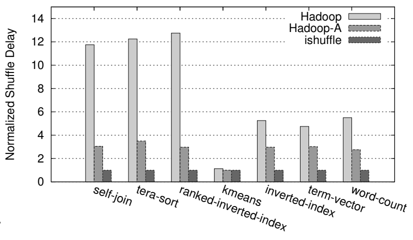


Figure 7: Shuffle delay due to three different approaches.

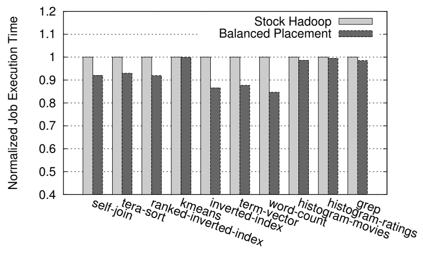


Figure 8: Performance with automated map output placement.

tively. iShuffle also outperformed Hadoop-A by 22.7%, 21.9%, and 21.1% in these benchmarks. The result with *k-means* benchmark does not show significant job execution time reduction between iShuffle and original Hadoop. This is because *k-means* only has 6 reduce tasks. With only one wave of reduce tasks, stock Hadoop was able to overlap the shuffle phase with map tasks and had similar performance as iShuffle. However, due to the additional delay of remote disk access, Hadoop-A had longer reduces, thus longer overall completion time.

Benchmarks like *inverted-index*, *term-vector*, and *wordcount* also fit in the shuffle-heavy category, but the shuffle volumes are smaller than other shuffle-heavy benchmarks. These benchmarks had less shuffle delay than other shuffle-heavy benchmarks simply because there was less data to be copied during the shuffle phase. Therefore, the performance improvement due to iShuffle was less. Figure 6 shows that iShuffle achieved 20.3%, 19.7%, and 15.6% better performance than stock Hadoop with these benchmarks, respectively. For these benchmarks, Hadoop-A still gained some performance improvement over stock Hadoop as the reduction on shuffle delay outweighed the prolonged reduce phase. However, the performance gain was marginal with 7.5%, 8.6%, and 5.5% improvement, respectively.

For the shuffle-light benchmarks, because the shuffle delay is negligible. Both iShuffle and Hadoop-A achieves almost no performance improvement. The performance degradation due to remote disk access in Hadoop-A is more obvious in this scenario.

We also compare the shuffle delay between the stock Hadoop, iShuffle, and Hadoop-A. Figure 7 shows the comparison of normalized shuffle delay. We used the shuffle delay of iShuffle as the based line. The results agree with the observation we made in previous experiments. iShuffle was able to reduce the shuffle delay significantly if the job had large volumes of shuffled data and multiple reduce waves. For benchmarks that have the largest shuffle-volume, the reductions in shuffle delay were more than 10x compared with stock Hadoop. For benchmarks with medium shuffle volume, the improvement on shuffle delay was from 4.5x to 5.5x. Figure 7 also suggests that iShuffle was on average 2x more effective in reducing shuffle delay than Hadoop-A.

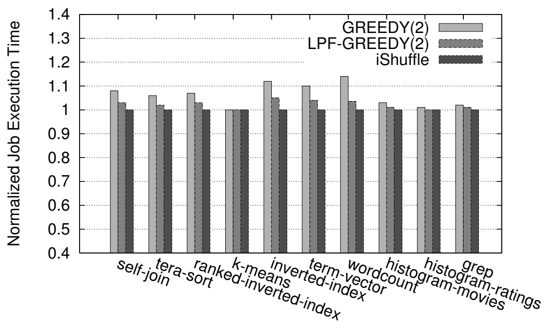


Figure 9: Performance with different placement balancing algorithms.

4.5 Balanced Partition Placement

We have shown that iShuffle effectively hides shuffle latency by overlapping map tasks and data shuffling. In this subsection, we study how the balanced partition placement affects job performance. To isolate the effect of partition placement, we first ran benchmarks under stock Hadoop and recorded dispatching history of reduce tasks. Then, we configured iShuffle to place partitions on nodes in a way that leads to the same reduce execution sequence. As such, job execution enjoys overlapped shuffle provided by iShuffle, but bears the same partitioning skew in stock Hadoop. We compare the performance with balanced partition placement and stock Hadoop.

Figure 8 shows the performance improvement due to balanced partition placement. The results show that iShuffle achieved 8-12% performance improvement over stock Hadoop. We attribute the performance gain to the prediction-based partition placement that mitigates the partitioning skew. It prevents straggler tasks from prolonging job execution time. The partition placement in iShuffle relies on accurate predictions of the individual

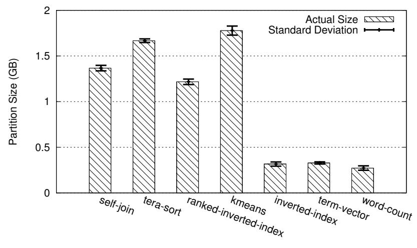


Figure 10: Accuracy of iShuffle partition size prediction.

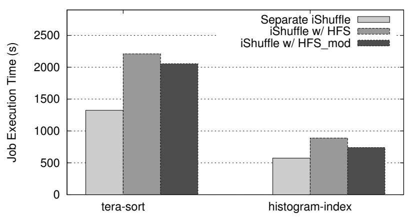


Figure 11: Performance of job mix of *tera-sort* and *histogram-movies*.

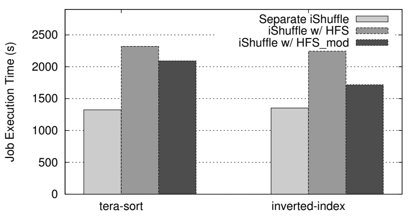


Figure 12: Performance of job mix of *tera-sort* and *inverted-index*.

partition sizes. Figure 10 shows the differences between measured partition sizes and the predicted ones. The results suggest that for all the shuffle-heavy benchmarks, iShuffle was able to estimate the final partition size with no more than 2% prediction errors.

4.6 Different Balancing Algorithms

In this subsection, we study how different partition balancing algorithms affect job performance. We compare our heuristic based partition balancing algorithm with two representative randomized balancing approaches.

`GREEDY(2)` implements the two-choice randomized load balancing algorithm proposed in [15]. It randomly picks up a map task for output placement and makes decision on which slave node to place the output using the two-choice greedy algorithm (i.e., `GREEDY(2)`). The node with less aggregated partition size (breaking ties arbitrarily) in the two randomly picked nodes is selected as the destination for the output placement. Different from `GREEDY(2)` which selects tasks randomly for placement, `LPF-GREEDY(2)` sorts tasks according to the descending order of their predicted partition sizes and always places tasks with larger partitions first (i.e., *largest partition first (LPF)*). Node selection is based on the two-choice randomized strategy.

Figure 9 compares the performance of different balancing algorithms. The results show that the simple heuristics used in iShuffle achieved $8-12\%$ shorter job completion time than `GREEDY(2)` in shuffle-heavy workloads (e.g., inverted-index). Since balanced partition placement is critical to job performance, `GREEDY(2)`'s randomization in task selection made it difficult to evenly distribute computation across nodes and contributed to the prolonged execution times. To confirm this, we ran `LPF-GREEDY(2)` with the same set of workloads. With the *largest partition first* heuristic in task selection, `LPF-GREEDY(2)` achieved close performance (on average only 2.5% longer runtimes) to iShuffle. In summary, although randomized balancing algorithms are easy to implement, the heuristics use in iShuffle is key to achieving balanced output placement.

4.7 Running Multiple Jobs

We further evaluate iShuffle in a multi-user Hadoop environment. We created multiple workload mixes, each contained two different MapReduce jobs. We ran one workload at a time with two jobs sharing the Hadoop cluster. We modified the Hadoop Fair Scheduler (HFS) (i.e., *iShuffle w/ HFS_mod*) to support runtime task-partition binding. For comparison, we also study the performance of iShuffle with the original HFS that enforces a minimum fair share on reduce tasks (i.e., *iShuffle w/ HFS*) and iShuffle running a single job on a dedicated cluster (i.e., *Separate iShuffle*).

The first experiment used the combination of a shuffle-heavy job and a shuffle-light job. Figure 11 shows the result of workload mix of *tera-sort* and *histogram-movies*. The results suggest that the modified HFS outperformed the original HFS by 16% and 25% for *tera-sort* and *histogram-movies*, respectively. Unlike the original HFS, which guarantees *max-min* fairness to jobs, iShuffle allows the reduce of one job to use more reduce slots. iShuffle prioritizes shuffle-light jobs because the execution time of their reduce tasks is short. Allowing shuffle-light jobs to run with more slots boosted their performance significantly. Although shuffle-heavy jobs suffered unfairness to a certain degree, their overall performance under the modified HFS was still better than that under the original HFS.

Next, we perform the experiment with two shuffle-heavy jobs. Figure 12 shows the performance of *tera-sort* and *inverted-index*. It shows that iShuffle improved job execution times by 8% and 23% over the original HFS in these two benchmarks. Although the size of input datasets of these two benchmarks are similar, *inverted-index* has a smaller shuffle volume. Therefore, its reduce tasks can be started earlier as their partitions required less time to shuffle. *tera-sort* had less improvement in this scenario because some of its reduce tasks are delayed by *inverted-index*. Table 2 shows more results of iShuf-

Table 2: Job completion time of co-running jobs.

Workload Mix	Stock Hadoop		iShuffle	
A + B	A	B	A	B
tera-sort+ grep	2210	1247	2144	1038
tera-sort+ histogram -ratings	2308	653	1976	530
tera-sort+term-vector	2576	2183	2349	1845
tera-sort+ wordcount	2341	1433	2126	1197
tera-sort+ k-means	1723	3764	3685	3748

fle with heterogeneous workloads compared with stock Hadoop. For most workload mixes with two jobs, iShuffle w/ modified HFS was able to reduce the job completion time for both jobs. The performance gain depends on the amount of shuffled data in these co-running jobs.

However, iShuffle had poor performance with workload mix *tera-sort* + *k-means*. We ran *tera-sort* with a 300GB dataset and *k-means* with a 15GB dataset. The result of *k-means* does not agree with previous observations for shuffle-light workloads. The co-running of *tera-sort* and *k-means* significantly degraded the performance of *tera-sort*. An examination of the execution trace revealed that although *k-means* has little data to exchange between map and reduce, it is compute intensive. iShuffle started *k-means* earlier than *tera-sort* and *k-means* occupied the reduce slots for a long time delaying the execution of *tera-sort*. The culprit was that for *k-means*, the partition size is not a good indicator of the execution time of its reduce tasks. Thus, iShuffle failed to balance the reduce workload on multiple nodes. A possible solution is to detect such outliers earlier and restart them on different nodes. Since such outliers often have small shuffle volume, the migration is not expensive.

5 Related Work

MapReduce is a programming model for large-scale data processing [7]. Hadoop, the open-source implementation of MapReduce, provides a software framework to support the distributed processing of large datasets [1].

There is great research interest in improving Hadoop from different perspectives. A rich set of research focused on the performance and efficiency of Hadoop cluster. Jiang *et al.* [10], conducted a comprehensive performance study of Hadoop, summarized the factors that can significantly improve the Hadoop performance. Verma *et al.* [17, 18], proposed cluster resource allocation approach for Hadoop. They focused on improving the cluster efficiency by minimizing resource allocations to jobs while maintaining their service level objectives. They estimated the execution time of a job based on its resource allocation and input dataset, and determined the minimum resource allocation for the job. Lama and Zhou [13] proposed and developed AROMA, a system that automates the allocation of Cloud resources and configuration of Hadoop parameters for achieving quality of service goals while minimizing the incurred cost. It uses a SVM-based approach to obtain the optimal job configuration. It adapts to ad-hoc jobs by robustly matching their resource utilization signature with previously executed jobs and making provisioning decisions accordingly.

A number of studies proposed different task scheduling algorithms to improve Hadoop performance. The Longest Approximate Time to End (LATE) scheduling algorithm [22] improved the job performance in heterogeneous environments. FLEX [20] is a scheduling algorithm that enforces fairness between multiple jobs in a Hadoop cluster. It optimized the performance of each job under different metrics. Zaharia *et al.*, proposed delay scheduling [21] as an enhancement to Hadoop Fair Scheduler. It exploited data locality of map task and significantly improved performance.

There are a few studies on skew mitigations. SkewReduce [11] alleviated the computational skew problem by applying a user-defined cost function on the input records. Partitioning across nodes relies on this cost function to optimize the data distribution. SkewTune [12] proposed a framework for skew mitigation. It repartitioned the long tasks to take the advantage of idle slots freed by short tasks. However, moving repartitioned data to idle nodes requires extra I/O operations.

Some recent work focused on the improvement of shuffle and reduce. MapReduce Online [6] proposed a push-based shuffle mechanism to support the online aggregation and continuous queries. MaRCO [3] overlaps the reduce and shuffle. But the early start of reduce generates partial reduces which could be the source of overhead for some applications. Hadoop Acceleration [19] proposed a different approach to mitigate shuffle delay and repetitive merges in Hadoop. It implemented a merge algorithm based on remote disk access and eliminated the explicit copying process in shuffle. However, this approach relies on the RDMA feature of Infiniband network, which is not available on commodity network hardware. Without RDMA, the remote disk access added significant overhead to reduce tasks. Moreover, Hadoop-A does not decouple shuffle and reduce, making it less effective for jobs with multiple reduce waves.

6 Conclusions

Hadoop provides a simplified implementation of the MapReduce framework, but its design poses challenges to attain the best performance in job execution due to tightly coupled shuffle and reduce, partitioning skew, and inflexible scheduling. In this paper, we propose *iShuffle*, a novel user-transparent shuffle service that provides optimized data shuffling to improve job performance. It decouples shuffle from reduce tasks and proactively pushes data to be shuffled to Hadoop node via a novel *shuffle-on-write* operation in map tasks. iShuffle further optimizes the scheduling of reduce tasks by automatic balancing workload on multiple nodes and runtime flexible reduce scheduling. We implemented iShuffle as a configurable plug-in in Hadoop and evaluated its effectiveness on a 32-node cluster with various workloads. Experimental results shows that iShuffle is able to reduce job completion time by as much as 30.2%. iShuffle also significantly improves job performance in a multi-user Hadoop cluster running heterogeneous workloads.

Acknowledgement

This research was supported in part by U.S. NSF CAREER Award CNS-0844983 and research grant CNS-1217979.

References

[1] Apache Hadoop Project. `http://hadoop.apache.org`.

[2] PUMA: Purdue mapreduce benchmark suite. `http://web.ics.purdue.edu/~fahmad/benchmarks.htm`.

[3] AHMAD, F., LEE, S., THOTTETHODI, M., AND VIJAYKUMAR, T. N. [inpress]mapreduce with communication overlap (marco). *Journal of Parallel and Distributed Computing* (2012).

[4] ANANTHANARAYANAN, G., AGARWAL, S., KANDULA, S., GREENBERG, A., STOICA, I., HARLAN, D., AND HARRIS, E. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (2011).

[5] CHIANG, R. C., AND HUANG, H. H. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[6] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. Mapreduce online. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2010).

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI)* (2004).

[8] DEWITT, D., AND GRAY, J. Parallel database systems: the future of high performance database systems. *Communication of ACM 35*, 6 (1992), 85–98.

[9] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1990.

[10] JIANG, D., OOI, B. C., SHI, L., AND WU, S. The performance of MapReduce: an in-depth study. *Proc. VLDB Endow.* (2010).

[11] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)* (2010).

[12] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skewtune: Mitigating skew in mapreduce applications. In *Proc. of the ACM SIGMOD* (2012).

[13] LAMA, P., AND ZHOU, X. AROMA: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)* (2012), pp. 63–72.

[14] LI, M., SUBHRAVETI, D., BUTT, A. R., KHASYMSKI, A., AND SARKAR, P. Cam: a topology aware minimum cost flow based resource manager for mapreduce applications in the cloud. In *Proc. of the ACM Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[15] MITZENMACHER, M. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California, Berkeley, 1996.

[16] TAN, J., MENG, X., AND ZHANG, L. Coupling scheduler for mapreduce/hadoop. In *Proc. of the ACM Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[17] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Aria: automatic resource inference and allocation for mapreduce environments. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)* (2011).

[18] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2011).

[19] WANG, Y., QUE, X., YU, W., GOLDENBERG, D., AND SEHGAL, D. Hadoop acceleration through network levitated merge. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[20] WOLF, J., RAJAN, D., HILDRUM, K., KHANDEKAR, R., KUMAR, V., PAREKH, S., WU, K.-L., AND BALMIN, A. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2010).

[21] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (2010).

[22] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

AutoPlacer: Scalable Self-Tuning Data Placement in Distributed Key-value Stores

João Paiva Pedro Ruivo Paolo Romano Luís Rodrigues
INESC-ID Lisboa / Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal
{joao.paiva,pedro.ruivo,paolo.romano,ler}@ist.utl.pt

Abstract

This paper addresses the problem of autonomic data placement in replicated key-value stores. The goal is to automatically optimize replica placement in a way that leverages locality patterns in data accesses, such that inter-node communication is minimized. To do this efficiently is extremely challenging, as one needs not only to find lightweight and scalable ways to identify the right data placement, but also to preserve fast data lookup. The paper introduces new techniques that address each of the challenges above. The first challenge is addressed by optimizing, in a decentralized way, the placement of the objects generating most remote operations for each node. The second challenge is addressed by combining the usage of consistent hashing with a novel data structure, which provides efficient probabilistic data placement. These techniques have been integrated in Infinispan, a popular open-source key-value store. The performance results show that the throughput of the optimized system can be 6 times better than a baseline system employing the widely used static placement based on consistent hashing.

1 Introduction

Distributed NoSQL key-value stores [10, 18] have emerged as the reference architecture for data management in the cloud. A fundamental design choice in these distributed data platforms is to select the algorithm used for determining the placement of objects (i.e., key/value pairs) among the nodes of the system. A data placement algorithm must simultaneously address two main, typically opposing, concerns: i) maximizing locality, by storing replicas of the data in the nodes that access them more frequently, while enforcing constraints on the object replication degree and on the capacity of nodes; ii) maximizing lookup speed, by ensuring that a copy of an object can be located as quickly as possible.

The data placement problem has been investigated in several alternative variants, e.g. [12, 16]. Classic approaches formulate the data placement problem as a constraint optimization problem, and use Integer Linear Programming techniques to identify the optimal placement strategy with the granularity of single data items. Unfortunately, these approaches suffer from several practical limitations. In first place, finding the optimal placement is a NP-hard problem, hence any approach that attempts to optimize the placement of each and every item is inherently non-scalable. Further, even if the optimal placement could be computed, it is challenging to maintain efficiently a (potentially very large) directory to store the mapping between items and storage nodes.

Directories are indeed used by several systems such as PNUTS [6] or BigTable [4]. To minimize the costs associated with the maintenance of the directory, these systems trade-off placement flexibility and support placement at a very coarse level, i.e. large data partitions rather than on a per instance basis. However, even if coarse granularity is used, the use of a directory service introduces additional round-trip delays along the critical execution path of data access operations, which can hinder performance considerably.

To avoid the above issues, many popular key-value stores, such as Cassandra [18], Dynamo [10], Infinispan [22], use random placement based on consistent hashing. By relying on random hash functions to determine the location of data across nodes, these solutions allow lookups to be performed locally, in an very efficient manner [10]. However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), solutions based on consistent hashing may result in highly sub-optimal data placements.

This paper presents AutoPlacer, a system aimed at self-tuning the data placement in a distributed key value store, which introduces a set of novel techniques to address the trade-offs described in the previous paragraphs. Unlike conventional solutions [12, 16], that formulate the data placement optimization problem as an intractable ILP problem, AUTOPLACER employs a lightweight self-stabilizing distributed optimization algorithm. The algorithm operates in rounds, and, in each round, it optimizes, in a decentralized fashion, the placement of the top-k "hotspots", i.e. the objects generating most remote operations, for each node of the system. This design choice has the advantage of reducing the number of decision variables for the data placement problem (solved at each round), ensuring its practical viability.

In order to be able to identify the "hotspots" of each node with low processing cost, AUTOPLACER adopts a state of the art stream analysis algorithm [23] that permits to track the top-*k* most frequent items of a stream in an approximate, but very efficient manner. The information provided by the Space-Saving Top-*k* algorithm is then used to instantiate the data placement optimization problem. We first study the accuracy of the solution from a theoretical perspective, deriving an upper bound on the approximation ratio with respect to a solution using exact frequencies. Next we discuss how to maximize the efficiency of the solution, showing how it can be made amenable for being partitioned in independent sub-problems, solvable in parallel.

Unlike solutions that rely on directory services, AUTOPLACER guarantees 1-hop routing latency. To this end, AUTOPLACER combines the usage of consistent hashing, which is used as the default placement strategy for less popular items, with a highly efficient, probabilistic mapping strategy that operates at the granularity of the single data item, achieving high flexibility in the relocation of (a possibly very large number of) hotspot items.

The key innovative solution introduced to pursue this goal is a novel data structure, which we named *Probabilistic Associative Array* (PAA). The goal of the PAA is to minimize the cost of maintaining a mapping associating keys with nodes in the system. PAAs expose the same interface of conventional associative arrays, but, in order to achieve space efficiency, they trade-off accuracy and rely on probabilistic techniques which can lead to inaccurate results with a user-tunable probability (these inaccuracies do not affect the correctness of the system, in worst case they may only degrade its performance). Internally, PAAs rely on Bloom Filters (BFs) and on Decision Tree (DT) classifiers. BFs are used to keep track of the elements inserted so far in the PAA in a space-efficient way; DTs are used to infer a compact set of rules establishing the associations between keys and values stored in the PAA. In order to maximize the effectiveness of the DT classifier, we expose a programmatic API that allows programmers to provide semantic information on the nature of the keys stored in the PAA (e.g., the data type of the value associated with the key). This information is then exploited, during the learning phase of the PAA's DT, to map keys into a multi-dimensional space that can be more effectively clustered by a DT classifier.

In summary, AUTOPLACER provides two key features:

- It introduces a novel iterative, decentralized, self-tuning data placement optimization scheme.
- It preserves efficient lookups, while achieving high flexibility in determining an optimized data placement, through the use of a new probabilistic data structure designed specifically for this purpose.

AUTOPLACER has been integrated in a popular, open-source key-value store, namely Infinispan: Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. The results show that AUTOPLACER can achieve a throughput 6 times better than a baseline system using consistent hashing.

The remaining of the paper is structured as follows. Our target system is characterized in Section 2. Section 3 provides a global overview of AUTOPLACER. Then, its components are described in more detail in the next two sections: the PAA internals are described in 4; a theoretical analysis of the optimizer's accuracy is provided in 5. Section 6 reports the results of the experimental evaluation of the system. Section 7 compares our system with related work. Finally, Section 8 concludes the paper.

2 System Characterization

The development of AUTOPLACER has been motivated by our experience [26, 25, 27] with the use of an existing, state-of-the art, key-value store, namely Infinispan [22] by Red Hat©. In Infinispan (and other similar products such as [18, 10]), data is stored in multiple nodes using consistent hashing. For each key, consistent hashing determines a *supervisor* node for that item. Items can be replicated. A node that stores a copy of data item *i* is denoted an *owner* of that item. Assume that *d* copies are maintained of each data item, the owners of data item *i* are deterministically assigned to be *j*'s supervisor plus its $d-1$ immediate successors (in the one hop distributed hash table that is used to implement consistent hashing).

Each node serves a dual purpose: it stores a subset of the data items maintained by the distributed store and also executes application code. The application code may be structured as a sequence of *transactions* (Infinispan supports transactional properties), with different isolation levels.

When the application code reads a data item, its value must be retrieved from one of its owners (which can be another node in the cluster). Thus, optimal performance is achieved if the node that executes a given application is the owner for the items it accesses more often. When the application writes a data item, all owners must be updated. Interestingly, the placement policy can also affect the performance of write operations. When multiple writes are performed in the context of a transaction, they can be applied in batch when the transaction commits. Hence, the larger the number of owners of keys updated by a transaction, the higher the number of nodes that have to be contacted during its commit phase.

Infinispan uses consistent hashing to ensure that all lookups can be executed locally. Unfortunately, in typical deployments of large-scale key-value stores, random data placement can be largely suboptimal as applications are likely to generate skewed access distributions [21], often dependent on the actual "type" of operations processed by each node [30, 9]. Also, workloads are frequently distributed according to load balancing strategies that strive to maximize locality [14]/minimize contention [2] in the data accesses generated by each node. As we will show in the evaluation section, all these facts make consistent hashing sub-optimal. Therefore, significant performance improvements can be achieved by using appropriate autonomic data placement strategies.

3 AUTOPLACER Overview

AUTOPLACER is designed to optimize data location in a decentralized manner, i.e., each node in the system contributes to the global optimization process. Since AUTOPLACER is aimed at systems that use consistent hashing as the default data placement policy, we also rely on consistent hashing to decentralize the optimization effort: each node is responsible for deciding the placement for the items it supervises. AUTOPLACER executes, cyclically, a sequence of optimization rounds. As a result of each round, a number of data items may be relocated. This happens only if the expected gains are above a minimum threshold. Each optimization round consists of the following sequence of six tasks.

Task 1: The first task of the AUTOPLACER approach consists of collecting statistics about the *hotspots* data, i.e., the top-k most accessed data items, at each node. In fact, instead of trying to optimize the placement of every data item in a single round, at each optimization round, AUTOPLACER only optimizes the placement of items that are identified as hotspots. Since this task is run cyclically, once some hotspots have been identified (and relocated) in a given round, new (different) hotspots are sought in the next round. Therefore, although in each round only a limited number of hotspots is identified, in the long run, a large number of data items may be selected over multiple optimization rounds, as long as gains can still be obtained from their relocation.

Task 2: The second task consists in having the nodes exchange statistics regarding the data items that were identified as hotspots during the current round. More precisely, each node gathers (from the remaining nodes of the platform) access statistics on any hotspot items it supervises.

Task 3: The above information is used in the third task (denoted the *optimization* task) to find an appropriate placement for those items. The result of this task is a *partial relocation map*, i.e., a mapping of where replicas of each hotspot items that the node supervises (for the current round) must be placed.

Task 4: Even if the number of hotspots tracked at each round is a small fraction of the entire set of items maintained in the key-value store, over multiple rounds the relocation map can grow in an undesired way, and may even be too large to be efficiently distributed to all nodes. This task is devoted to encoding the relocation map in a probabilistic data structure that can be efficiently replicated on all nodes in order to ensure fast lookups, i.e. a *Probabilistic Associative Array* (PAA). Specifically, each node computes the PAA for the (relocated) objects it supervises.

Task 5: Once each PAA has been computed, each node disseminates it among all nodes. By assembling the PAAs received from all the nodes in the system, each node can locally build an object lookup table that includes updated information on the placement of data optimized during this round.

Task 6: Finally, at the end of each round, the data items for which new locations have been derived are transferred (using conventional state-transfer facilities [15, 28]) in order to match the new data placement.

As can be inferred from the previous description, the work is divided among all nodes and communication takes place only during tasks 2, 5, and 6, in order to, respectively, exchange statistical information on hotspots, distribute the PAA, and finally relocate the objects. Also the tasks that require communication are performed in parallel, without the help of any centralized component.

In the next subsections we provide more information about the two main components of AUTOPLACER, namely, the optimizer (executed by Task 3) and the PAA (built in Task 4 and used subsequently to perform data lookups locally).

3.1 Optimizer

Most works, e.g., [30, 20, 16, 12], in the area of data placement (and of its many variants [16, 12]) assume that the objective and constraint functions of the optimization problem can be expressed (or approximated) via linear functions, and accordingly formulate an Integer Linear Programming (ILP) problem. The ILP model can indeed

$\mathcal{N}$	the set of nodes j in the system
$\mathcal{O}$	the set of objects i in the system
X	a binary matrix in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise
r_{ij}, w_{ij}	the number of read, resp. write, accesses performed on a object i by node j
cr^r, cr^w	the cost of a remote read, resp. write, access
cl^r, cl^w	the cost of a local read, resp. write, access
d	the replication degree, that is number of replicas of each object in the system
S_j	the capacity of node j.

Table 1: Parameters used in the ILP formulation.

be adopted also for the specific data placement problem tackled in this paper. To this end, one can model the assignment of data to nodes by means of a binary matrix X, in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise. Further, one can associate (average, or per object) costs with local/remote read/write operations. The ILP problem is then formulated as the minimization of the objective function that expresses the total cost of accessing all data items across all nodes, subject to two constraints: i) the number of replicas of each object must meet a predetermined replication degree, and ii) each node has a finite capacity (it must not be assigned more objects than it can store). In Table 1 we list the parameters used in the problem formulation, which aims at minimizing the following cost function:

$$\sum_{j \in \mathcal{N}} \sum_{i \in \mathcal{O}} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij}) \quad (1)$$

subject to:

$$\forall i \in \mathcal{O} : \sum_{j \in \mathcal{N}} X_{ij} = d \wedge \forall j \in \mathcal{N} : \sum_{i \in \mathcal{O}} X_{ij} \leq S_j$$

Despite its convenient mathematical formulation, ILP problems are NP-hard. Further, solving the above ILP problem would require to collect and exchange among nodes access statistics for all objects in the system. We tackle these drawbacks by introducing a lightweight, multi-round distributed optimization algorithm, which we describe in the following.

3.1.1 Space-Saving Top-k algorithm

An important building block of AUTOPLACER is the Space-Saving Top-k algorithm by Metwally *et al.* [23]. This algorithm is designed to estimate the access frequencies of the top-k most popular objects in an approximate, but very efficient way, i.e. by avoiding maintaining information on the access frequencies (namely counters) for each object in the stream. Conversely, the Space-Saving Top-k algorithm algorithm maintains a tunable, constant, number m, where $m \ll |\mathcal{O}|$, of counters, which makes it extremely lightweight. On the downside, the information returned in the top-k list may be inaccurate in terms of both the elements that compose it and their estimated frequency. However, this algorithm has a number of interesting properties concerning the inaccuracies it introduces. First, it ensures that the access frequencies of the objects it tracks are always consistently overestimated. Also, its maximum overestimation error is known, and is equal to the frequency of the least frequently accessed item present in top-k, denoted as F_k. Finally, its space-requirements can be tuned to bound the maximum error introduced in the frequency tracking, as we will further discuss in Section 5.

3.1.2 Using Approximate Information

In AUTOPLACER each node j runs 2 distinct instances, noted as $top\text{-}k_j^{rd}$, resp. $top\text{-}k_j^{wr}$, of the Space-Saving Top-k algorithm, used to track the k most frequently read, resp. updated, data items during the current optimization round. We denote with $top\text{-}k_j(\mathcal{O})$ the subset of cardinality k (of the entire data set $\mathcal{O}$) contained in both the read and write top-k instances at node j, and with $top\text{-}K(\mathcal{O}) = \cup_{j \in \mathcal{N}}(top\text{-}k_j(\mathcal{O}))$ the union of the top-k data items across all nodes.

By restricting the optimization problem to the top-k accessed data items we reduce the number of decision variables of the ILP problem significantly, namely from $|\mathcal{O}||\mathcal{N}|$ to $O(k|\mathcal{N}|)$ (where $k \ll |\mathcal{O}|$). This choice is crucial to guarantee the scalability of the proposed approach. However, it requires to deal with the incomplete and approximate nature of the data (read/write) access statistics provided by the top-k algorithm, which we denote with $\hat{r}_{ik}$,$\hat{w}_{ik}$ to distinguish them from their exact counterparts (r_{ik},w_{ik}). Also, we use the notation $\hat{X}$ to refer to the solution of the optimization problem using as input the access statistics provided by the top-k algorithm, and distinguish it from the one obtained using the exact access statistics in input, which we denote X^{opt}.

A first problem to address is related to the possibility of lacking information concerning the access frequency by some node j for some data item $i \in top\text{-}K(\mathcal{O})$: this can happen in case i has not been tracked in $top\text{-}k_j(\mathcal{O})$, but is present in the $top\text{-}k_{j'}(\mathcal{O})$ of some other node $j' \neq j$. To address this issue, we simply set to 0 the frequencies $\hat{r}_{ij}$,$\hat{w}_{ij}$

Finally, the approximate nature of the information provided by the Space-Saving Top-k algorithm may impact the quality of the identified solution. A theoretical analysis aimed at evaluating this aspect will be provided in Section 5.

3.1.3 Accelerating the solution of the optimization problem

To accelerate the solution of the optimization problem we take two complementary approaches: relaxing the ILP problem, and parallelizing its solution.

The ILP problem requires decision variables to be integers and is computationally onerous [30]. Therefore, we transform it into an efficiently solvable linear programming (LP) problem. To this end, we let the matrix $\hat{X}$ assume real values between 0 and 1 (adding an extra constraint $\forall i \in \mathcal{O}, \forall j \in \mathcal{N}\ 0 \leq \hat{X}_{ij} \leq 1$). Note that the solutions of the LP problem can have real values, hence each object is assigned to the d nodes which have highest $\hat{X}_{ij}$ values. As in [30], we use a greedy strategy according to which, if the assignment to a node causes a violation of its capacity constraint, the assignment is iteratively attempted to the node that has the $d+k$-th ($k \in [1, |\mathcal{N}| - d]$) highest scores.

Second, we introduce a controlled relaxation of the capacity constraint, which allows us to partition the ILP problem into $|\mathcal{N}|$ independent optimizations problems that we solve in parallel across the nodes of the platform. Let $top\text{-}k_j(\mathcal{O}|n)$ be the set of keys in $top\text{-}k_j(\mathcal{O})$ of node j that node n supervises. Each node j sends its $top\text{-}k_j(\mathcal{O}|n)$ to each other node n in the system. As a result each node j also gathers the access statistics $top\text{-}K(\mathcal{O}|j) = \cup_{n \in \mathcal{N}}\ top\text{-}k_n(\mathcal{O}|j)$ concerning the current hotspots that j supervises. At this point each node j computes the new placement for the data in $top\text{-}K(\mathcal{O}|j)$.

Note that since we are instantiating the (I)LP optimization problems in parallel, and in an independent fashion, we need to take an additional measure to guarantee that the capacity constraints are not violated. To this end we instantiate the (I)LP problems at each node j with a capacity $S'_j = S_j - |\mathcal{N}|k$. In practice, this relaxation is expected to have minimum impact on the solution quality as $k \ll S_j$.

Overall, at the end of an optimization round each node j produces two outputs: the partial relocation map $\hat{X}$, and the cost reduction achievable by relocating the data in $top\text{-}K(\mathcal{O}|j)$ according to $\hat{X}$, which we denote as Δ_{C_j}. Δ_{C_j}, which is computed on the basis of Equation 1, allows estimating the gain achievable by performing this optimization round, and, as we will discuss shortly, is used in AUTOPLACER to determine the completion of the round-based optimization algorithm.

3.2 Probabilistic Associative Array: Abstract Data Type Specification

Even though in each round AUTOPLACER optimizes the placement of a relatively small number of data items, over multiple optimization rounds the number of relocated objects can grow very large. Hence, a relevant issue is related to the overhead for maintaining, and replicating, a possibly very large relocation map. Indeed the relocation map can be seen as an associative array in which each entry is a pair mapping a data item to the set of nodes that own it.

Method	Input Parameters	Output
CREATE	Set⟨Key,Value[d]⟩, α, β	PAA
GET	Key	Value[d]
ADD	Set⟨Key,Value[d]⟩	PAA
GETDELTA	PAA	ΔPAA
APPLYDELTA	ΔPAA	PAA

Table 2: PAA Interface.

The Probabilistic Associative Array (PAA) is a novel data structure that allows maintaining an associative array in a space efficient, but approximate way. We present the PAA as an abstract data type, with an interface analogous to conventional associative arrays. Later in Section 4, we will discuss how it has been implemented in AUTOPLACER.

The PAA is characterized by the API reported in Table 2, which is similar to that of a conventional associative array, including methods to create and query a map between keys and (constant d-sized) arrays of values. To this end, the PAA API includes three main methods: the CREATE method, which returns a new PAA instance and takes as input a set of pairs in the domain (key $\times$ array[d] of values) to be stored in the PAA (called, succinctly, *seed map*) and two tunable error parameters α and β (discussed below); the GET method, which allows querying the PAA obtaining the array of values associated with the key provided as input parameter, or $\perp$ if the key is not contained in the PAA; the ADD method, which takes an input a seed map and adds it to an existing PAA.

The PAA trades accuracy for space efficiency, and may return inaccurate results when queried. In the following we specify the properties ensured by the GET method of a PAA:

• it may provide *false positives*, i.e., to provide a return value different from $\perp$ for a key that was not inserted in the PAA. The probability of false positives occurring is controlled by parameter α.

• it has no *false negatives*, i.e., it will never return $\perp$ for a key contained in the seed map.

• it may return an *inaccurate* array of values for a key contained in the seed map. The probability of returning inaccurate arrays is controlled by parameter β. In other words, with some small and controlled probability, the data items may be located in different nodes than those specified by the seed map (thus, the efficiency of lookup may cause some degree of sub-optimal placement).

• its response is deterministic, i.e., for a given instance of a PAA, the return value for any given key is always the same.

Finally, the PAA API contains two additional methods that allow to update the content of a PAA in an incremental fashion: GETDELTA, and APPLYDELTA. GETDELTA takes as input a PAA and returns an encoding, denoted as ΔPAA, of the differences between the base PAA over which the method is invoked (say PAA_1) with respect to the input PAA, say PAA_2. The ΔPAA returned by GETDELTA can then be used to obtain PAA_2 by invoking the method APPLYDELTA over PAA_1 and passing as input parameter ΔPAA.

3.3 The AUTOPLACER iterative algorithm

We now provide, in Algorithm 1, the pseudo-code formalizing the behavior of the AUTOPLACER algorithm executed by a node j. Each node maintains a local lookup table, denoted as *LookupT*, that consists of an array of PAAs, one per each node j in the system. Specifically, j's entry of *LookupT* is used to keep track of the objects supervised by node j that have already been relocated by AUTOPLACER. For any given round, *LookupT* is the same on all nodes.

At the beginning of each round, j collects statistics concerning its top-k most frequently read/written data items. This activity is encapsulated by the `collectStats` procedure, which is designed to track only accesses to objects whose placement had not been previously optimized in previous rounds. This measure is necessary, as, otherwise, in presence of stable distributions of the data access among nodes (i.e., stable workloads), the top-k lists at each node may quickly stagnate. Especially in case of skewed distributions the top-k lists would tend to track the very same objects (i.e., the most popular ones) along every round.

By tracking only the keys whose placement has not been optimized in previous rounds, it is guaranteed that, in two different rounds, two disjoint set of objects are considered by the optimization algorithm, leading to the analysis of progressively less "hot" data items. Further, it prevents the possibility of ping-pong phenomena [13], i.e. the continuous re-location of a key between nodes, as it guarantees that the position of each object is optimized at most once.

To determine whether an access to a data item should be traced or not, the `collectStats` procedure is provided with *LookupT* as input (we recall that *LookupT* keeps track of all items whose placement has been previously optimized). Upon a read/write access on a data item, the `collectStats` procedure, whose code is not reported for space constraints, checks if the item is contained in *LookupT* and, in the positive case, it avoids

```
1  Array[1...|$\mathcal{N}$|] of PAA : LookupT={$\perp,\ldots,\perp$};
2  PAA: tmpPAA=$\perp$;
3  do
4      Array[1...|$\mathcal{N}$|] of Set$\langle i,r,w\rangle$ : req=$\perp$;
5      $\langle top_k^{rd}, top_k^{wr}\rangle \leftarrow$ collectStats(LookupT);
6      foreach $n \in \Pi$ do
7          send({$\langle i,r,w\rangle \in \{top_k^{rd} \cup top_k^{wr}\}$ such that
               supervisor($i$) $= n$}) to $n$;
8      foreach $n \in \Pi$ do
9          req[j]$\leftarrow$ receive() from $n$;
10     $\langle \hat{X}, \Delta_{C^j}\rangle \leftarrow$ Optimize(req);
11     tmpPAA $\leftarrow$ LookupT[$j$];
12     tmpPAA.ADD($\hat{X}$);
13     ΔPAA: delta $\leftarrow$ tmpPAA.GETDELTA(LookupT[$j$]);
14     broadcast(delta,$\Delta_{C^j}$);
15     $\Delta_{C^*} \leftarrow 0$;
16     foreach $n \in \Pi$ do
17         [delta,$\Delta_{C^n}$] $\leftarrow$ receive() from $n$;
18         LookupT[$n$]$\leftarrow$LookupT[$n$].APPLYDELTA(delta);
19         $\Delta_{C^*} \leftarrow \Delta_{C^*} + \Delta_{C^n}$;
20     moveData();
21 while $\Delta_{C^*} > \gamma$;
```

Algorithm 1: AUTOPLACER's behavior at node j

tracing this access. Notice that we are assuming that the data access frequencies do not change significantly during the entire optimization process. Extending AUTOPLACER to cope with scenarios in which applications' data access patterns change at a frequency higher than AUTOPLACER's complete optimization cycle is outside of the scope of this paper and will be subject of future work (see Section 8).

Next the nodes exchange the information collected by `collectStats`. Since we also parallelize the optimization procedure, we send to each node only the statistical information that will be relevant to the computation that will be performed at that node, i.e., the statistical information regarding the data items it supervises.

At this point, each node optimizes the placement for the objects it supervises (primitive `Optimize`), determining their new owners (encoded in the partial relocation map, denoted $\hat{X}$). The node also computes the reduction of the local cost function (denoted as Δ_{C^j}) that the new assignment brings.

Then, node j computes a temporary PAA, based on the previous value of its PAA (stored in *LookupT*[j]) and on the new additional relocation information $\hat{X}$ (lines 12-13). The API of the PAA is then used to extract the relevant deltas from the existing PAA that need to be disseminated, in order to avoid sending the entire PAA again (line 14). These deltas are exchanged among nodes, and applied locally, such that every node can update all entries of *LookupT* (lines 17-20).

Each optimization round ends by triggering the relocation of the data via the `moveData()` primitive. This primitive will use the updated PAAs to determine the set of items that have been re-located, and gives the necessary commands to perform the corresponding state transfers. Several state transfer techniques could be used for this purpose [15, 28], whose complexity is dependant on the consistency guarantees that the key-value store implements (e.g. transactional vs eventual consistency). These mechanisms are indeed orthogonal to the AUTOPLACER system.

Finally, AUTOPLACER relies on a simple self-stabilizing mechanism that halts the distributed optimization algorithm if the "gain" achieved during the last optimization round does not exceed a user-tunable minimum threshold, denoted γ (line 22). This allows avoiding to analyze the "tail" of the data access distribution, whose optimization would lead to negligible gains. We chose as metric to evaluate the optimization gain the reduction of the cost function achieved during the last optimization round, ΔC^*. To compute this value, each node j disseminates the value for the reduction of its local cost function ΔC_j along with delta, in line 15. At the end of this dissemination phase each node of the system can deterministically compute ΔC^* and evaluate the predicate on the termination of the optimization algorithm.

3.4 The lookup function

Algorithm 2 shows the pseudocode for the lookup function for a key k. First, consistent hashing is used to identify the supervisor of k, say s. We then check whether the PAA associated with s contains k. In the positive case, we use the mapping information provided by *LookupT*[s] to identify the set of nodes that are currently maintaining key k. Otherwise, we simply return the set of owners for k as determined by consistent hashing (d is the replication degree).

```
Array[1...d] of Nodes LOOKUP(Key k)
    if LookupT[supervisor(k)].GET(k) ≠ ⊥ then
        return LookupT[supervisor(k)].GET(k);
    else
        s ← supervisor(k);
        return {s, s+1, ..., s+d-1};
```

Algorithm 2: PAA-based lookup function

4 Probabilistic Associative Array Internals

4.1 Building Blocks

Scalable Bloom filters (SBF) [1] are a variant of Bloom filters (BF) [3], a well know data structure that supports probabilistic test for membership of elements in sets. A BF never yields false negatives (if the query returns that an element was not inserted in a BF, this is guaranteed to be true). However, a BF may yield false positives (a query may return true for an element that was never inserted) with some tunable probability α, which is a function of the number of bits used to encode the BF and of the number of elements that one stores in it (that must be known a-priori). SBFs extend BFs in that they can adapt their size dynamically to the number of elements effectively stored, while still ensuring a bounded false positive probability. This is achieved by creating, on demand, a sequence of BFs with increasing capacity.

VFDT [11] is a classifier algorithm that induces decision trees over a stream of data, i.e. without assuming the a-priori availability of the entire training data set unlike most existing decision trees [24]. VFDT is an incremental online algorithm, given that it has a model available at any time during its run and refines the model over time (by performing new splits, or pruning unpromising leaves) as it is presented with additional training data. As classical off-line decision trees, the output of VFDT is a set of rules that allows to map a point in the feature space to a target discrete class.

The PAA uses SBFs and VFDT in the following manner. SBFs are used to assert if a key was stored in the PAA. VFDT is used to obtain the set of values associated with a key stored in tha PAA. The next paragraphs explain how this technique works in detail.

4.2 FeatureExtractor Key Interface

In order to maximize the effectiveness of the machine-learning statistical inference, programmers can optionally provide semantic information on the type of key inserted in a PAA, by having their keys implementing the FEATUREEXTRACTOR interface. This interface exposes a single method, GETFEATURES(), which returns a set of pairs $\langle$*featureName*, *featureValue*$\rangle$, where *featureName* is a unique string identifying each feature and *featureValue* is a (continuous or discrete) value defining the value of that feature for the key.

The purpose of this interface is to allow a key to be mapped, in a semantically meaningful (and hence inherently application dependant) way, into a multi-dimensional feature-space that can be more efficiently analyzed and partitioned by a statistical inference tool. Features can be "naturally" derived from the data model used in the application. For instance, if an object-oriented (or relational model) is used, a typical encoding for the key corresponding to an object of class "Person" with ID=3 may be "Person-3". The FEATURE-EXTRACTOR interface can then simply parse the key and return the pair $\langle$"Person", $id\rangle$. This can be further illustrated considering the real example of the TPC-C benchmark, which we used in our evaluation. In this case, a "Customer" object with id c_1 would be associated with a feature, $\langle$"Customer", $c_1\rangle$. Further, since in TPC-C a customer is statically registered in a "Warehouse" object, c_1 would have a second feature $\langle$"Warehouse", $w_1\rangle$, being w_1 the identifier of the warehouse where c_1 is registered. Hence, a different customer c_2 registered with a warehouse w_2 would be associated with the features $\langle$"Customer", $c_2\rangle$ and $\langle$"Warehouse", $w_2\rangle$, while the object representing warehouse w_2 itself would be associated with the features $\langle$"Customer", N/A$\rangle$ and $\langle$"Warehouse", $w_2\rangle$.

Note that this sort of feature extraction can be easily automated, provided the availability of information on the mapping between the application's domain model, in terms, e.g., of entities and relationships, and the underlying key/value representation.

4.3 PAA Operations

• CREATE: a SBF is created, sizing it to ensure the target error rate α and populating it with the keys passed as input parameter. Further we train d new instances of VFDT. The i-th instance of VFDT ($i \in [1,\ldots,d]$) is trained by using a dataset containing, for each key k in the seed map, an entry composed by the mapping of k in the feature space (obtained using k's FeatureExtractor interface), and as target class value, the i-th value associated with k in the seed map. As we are creating a decision-tree from scratch over a fully-known training set, we use in this phase VFDT as an offline-learner. This allows us to tightly control the cardinality of the rule set it generates to achieve arbitrary accuracy in encoding the mapping, and hence fine tune the pruning of the rule set to achieve the user specified parameter β.

• GET: queries for a key k are performed by first querying the SBF. If the response is negative, $\perp$ is returned. Otherwise (and this may be a false positive with probability α), the VFDT is queried by transforming k in its representation in the feature space by means of the FeatureExtractor interface. If k had actually been inserted in the PAA, the query to the SBF is guaranteed to return a correct result. However, it may still be wrongly classified by the VFDT, which may return any of the target classes that it observed during the training phase.

• ADD: to implement this method, we leverage on the incremental features of the SBF and VFDT. To this end, we first insert each of the entries k passed as input parameter into the SBF. This may lead to the generation of an additional, internal bloom filter, to ensure that the bound on α is ensured. Next we incrementally train the VFDT instances currently maintained in the PAA, by providing them, in a single batch, the entire set of key/value pairs that is being added to the PAA. In this phase we control the learning of the new mapping in a single batch, by allowing the VFDT algorithm to scan the new training set multiple times until we reach the target bound on misclassification β is satisfied.

• GETDELTA: the output consists of the binary diff of the SBFs, plus the rule set of the VFDT maintained by the PAA over which this method is invoked.

• APPLYDELTA: symmetrically to what is done in GETDELTA, this method generates a new PAA, whose SBF is obtained by applying the binary SBF diff contained in the input ΔPAA to the SBF of the PAA over which this method is invoked. The rule set of the output PAA is set equal to the one contained in the input ΔPAA.

5 Optimizer Analysis

As already noted, the approximate nature of the information provided by top-k may affect the quality of the identified solution. An interesting question is therefore how degraded is the quality of the data placement solution when using top-k. In the following theorem we provide an answer to this question by deriving an upper bound on the approximation ratio of the proposed algorithm in an optimization round. Our proof shows that the approximation ratio is a function of the maximum approximation error provided by any $top\text{-}k_j(\mathscr{O})$, which we denote e^*, and of the average frequency of access to remote data items when using the optimal solution.

Theorem 1 The approximation ratio of the solution $\hat{X}$ found using the approximate frequencies $\hat{r}_{ik}$,$\hat{w}_{ik}$ is:

$$1+\frac{d}{|\mathscr{N}|-d}\phi, \text{ with } \phi=\frac{e^*(cr^r+cr^w)}{cr^r rR+cr^w rW}$$

where e^* is the maximum overestimation error of top-k, and rR, resp. rW, is the average, across all nodes, of the number of read, resp. write, remote data items using the optimal data placement X^{Opt}.

Proof Let us now denote with $C(X,r_{ij},w_{ij})$ the cost function used in Eq. 1 of the ILP formulation restricted to the data items contained in $top\text{-}K(\mathscr{O})$:

$$\sum_{j\in\mathscr{N}}\sum_{i\in top\text{-}K(\mathscr{O})}\overline{X}_{ij}(cr^r r_{ij}+cr^w w_{ij})+X_{ij}(cl^r r_{ij}+cl^w w_{ij})$$

and with $Opt = C(X^{opt}, r_{ij}, w_{ij})$ the value returned by the cost function using the binary matrix X^{opt} obtained solving the ILP problem with exact access statistics r_{ij}, w_{ij}.

Let lR, resp. rR, be the average, across all nodes, of the number of read accesses to local, resp. remote, data items using the optimal data placement X. lW and rW are analogously defined for write accesses. These can be directly computed, once known X^{Opt} and r_{ij},w_{ij} as:

$$rR = \frac{\sum_{j\in\mathscr{N}} \sum_{i\in top\text{-}K(\mathscr{O})} \overline{X}_{ij}^{Opt} r_{ij}}{|top\text{-}K(\mathscr{O})|(|\mathscr{N}|-d)}$$

$$rW = \frac{\sum_{j\in\mathscr{N}} \sum_{i\in top\text{-}K(\mathscr{O})} \overline{X}_{ij}^{Opt} w_{ij}}{|top\text{-}K(\mathscr{O})|(|\mathscr{N}|-d)}$$

$$lR = \frac{\sum_{j\in\mathscr{N}} \sum_{i\in top\text{-}K(\mathscr{O})} X_{ij}^{Opt} r_{ij}}{|top\text{-}K(\mathscr{O})|d}$$

$$lW = \frac{\sum_{j\in\mathscr{N}} \sum_{i\in top\text{-}K(\mathscr{O})} X_{ij}^{Opt} w_{ij}}{|top\text{-}K(\mathscr{O})|d}$$

We can then rewrite Opt and derive its lower bound:

$$\begin{aligned} Opt =& |top\text{-}K(\mathscr{O})|((|\mathscr{N}|-d)(cr^r rR + cr^w rW) + \\ &+ d(cl^r lR + cr^w lW)) \geq \\ &\geq |top\text{-}K(\mathscr{O})|(|\mathscr{N}|-d)(cr^r rR + cr^w rW) \end{aligned} \quad (2)$$

Next, let us derive an upper bound on the "error" using the solution $\hat{X}$ obtained instantiating the ILP problem using the top-k-based frequencies $\hat{r}_{ij}, \hat{w}_{ij}$. The worst scenario is that an object $o \in \mathscr{O}$ is not assigned to the d nodes that access it most frequently because they do not include o in their top-k. In this case we can estimate the maximum frequency with which o can have been accessed by any of these nodes as e^*. Hence if we evaluate the cost function $C(\hat{X}, r_{ij}, w_{ij})$ using the exact data access frequencies, and the solution $\hat{X}$ of the ILP problem computed using approximate access frequencies, we can derive the following upper bound:

$$C(\hat{X}, r_{ij}, w_{ij}) \leq Opt + |top\text{-}K(\mathscr{O})| d e^* (cr^r + cr^w) \quad (3)$$

The approximation ratio is therefore:

$$\frac{C(\hat{X}, r_{ij}, w_{ij})}{C(X^{Opt}, r_{ij}, w_{ij})} \leq 1 + \frac{d}{(|\mathscr{N}|-d)} \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW} \quad (4)$$

In the following corollary we exploit the bounds on the space complexity of the Space-Saving Top-k algorithm [23] to estimate the number of distinct counters to use to achieve a target approximation factor $1 + \frac{d}{|\mathscr{N}|-d}\phi$.

Corollary 2 The number m of individual counters maintained by the Space-Saving Top-k algorithm, to achieve an approximation factor equal to $1 + \frac{d}{|\mathscr{N}|-d}\phi$ is:

$$m = \frac{SL}{\phi} \frac{cr^r rR + cr^w rW}{cr^r + cr^w}$$

where SL is the total number of accesses in the stream.

Proof Derives from Theorem 6 of the work that introduced the Space-Saving Top-k algorithm [23], which proves that to guarantee that the maximum overestimation error $e^* \leq \varepsilon F_k$, where F_k is the frequency of the k-th element in top-k, it is sufficient to use $m = \frac{SL}{\varepsilon F_k}$ counters.

Finally, since in each round AUTOPLACER optimizes the placement of a disjoint set of items, it follows that, if we assume stable data access distributions, the approximation ratio achieved by the optimization algorithm during round i will necessarily be lower (hence better) than for round $i-1$. In fact, at each round, the frequencies of the items tracked by the top-k will be lower than in the previous rounds, and, consequently, e^* will not increase over time.

6 Evaluation

In order to evaluate experimentally AUTOPLACER, we integrated it in the Infinispan key-value store. As benchmarking platform, we have used a cluster of 40 virtual machines (deployed on 10 physical machines) running Xen, equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet. Since Infinispan provides support for transactions, we developed for our experimental study a porting of a well-known benchmark for transactional systems, namely the TPC-C benchmark [21], which we adapted to execute on a key-value store[1]. This choice is motivated by the fact that TPC-C is a complex benchmark, which generates workloads representative of realistic OLTP environments, with complex and heterogeneous transactions having very skewed access patterns and high conflict probability. This is in contrast with common key-value store benchmarks [7], which are typically composed of simple synthetic workloads.

Since our evaluation focuses on assessing the effectiveness of AUTOPLACER in different scenarios of locality, we have modified the benchmark such that we can induce controlled locality patterns in the data accesses of each node. This modification consists in configuring the benchmark such that the transactions originated on a given node access with probability p data associated with a given warehouse, and with probability $1-p$ data associated a warehouse chosen randomly. So, for example, by setting $p = 90\%$, nodes will have disjoint data access patterns (each accessing a different warehouse) for 90% of the transactions, while the remaining 10% access data uniformly.

[1]The code of AUTOPLACER and of the porting of TPC-C employed in this evaluation study are freely available in the Cloud-TM project public repository: `http://github.com/cloudtm`

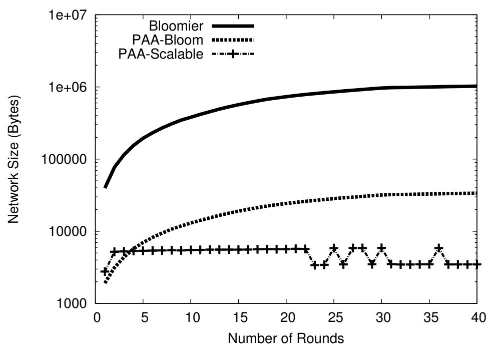


Figure 1: Traffic generated by a node using different associative arrays.

Mechanism	Re-located objects	Local space (KB)
PAA-scalable	26600	150.8
PAA-Bloom	26600	31.84
Bloomier	26600	575.3

Table 3: Re-located objects and size of different PAA implementations

6.1 Probabilistic Associative Array

In this section we study tradeoffs in the space efficiency and accuracy involved in the configuration and implementation of the PAA. For these results, we have configured the benchmark with 100% locality. In order to use the PAA with TPC-C, we modified the TPC-C keys in order to implement the Feature Extractor Interface according to the static attributes of the objects they represent.

Bloom Filter Figure 1 presents the network bandwidth of different implementations of the PAA, compared with another form of probabilistic associative arrays, the Bloomier Filters (BLOOMIER) [5] as the rounds advance in the system. One implementation of the PAA uses regular Bloom filters (PAA-BLOOM) [3], while the other uses a scalable Bloom filter (PAA-SCALABLE) [1]. Both PAAs were configured with $\alpha = \beta = 0.01$, and the Bloomier filter's false positive probability was also set to 0.01. We note that the best solution is the one that allows to propagate in the network only differential updates with regard to the previous state, i.e, the PAA-scalable.

Table 3 shows the correspondent local storage requirements at the end of the experiment. As it can be seen, PAA-SCALABLE has higher storage requirements than PAA-BLOOM. This is unsurprising, as scalable bloom filters are known to achieve lower compression than traditional bloom filters when fed with the same data set and configured to yield the same false positive rates [1].

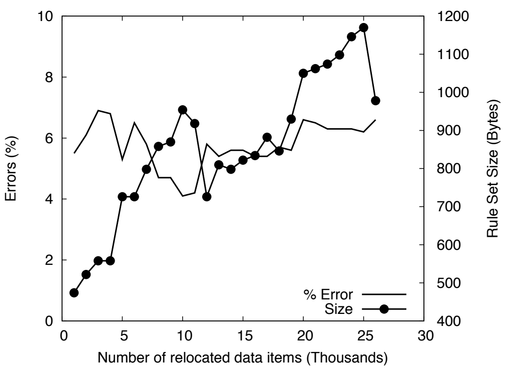


Figure 2: Error probability and rule set size for VFDT

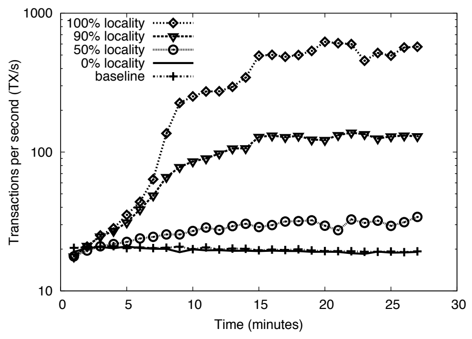
However, the storage requirements of both solutions are still considerably smaller than those of Bloomier filters.

Machine Learner Figure 2 presents the error probability and space required by the DT to encode the objects moved in every round of the experiment. As more objects are moved in the system, the number of rules increases, leading to an increase in the size taken by this portion of the PAA. However, the machine learner can represent the mapping of 26600 keys in 1000 Bytes, which correspond to 213 rules. Furthermore, it can also be observed that while a significant number of keys is added to the machine learner (around 1000 per round), the error remains relatively stable.

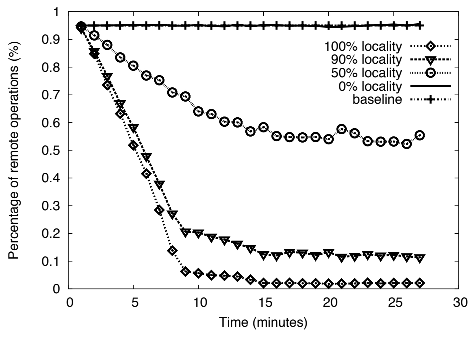
6.2 Leveraging from Locality

This section shows how AUTOPLACER is able to leverage form locality patterns in the workload. The results were obtained with TPC-C, adapted as explained before and with a replication of degree $d = 2$.

Figure 3(a) shows the throughput of AUTOPLACER, compared with the non-optimized system using consistent hashing for different degrees of locality in the workload. In the baseline system, no matter how much locality exists in the workload, since consistent hashing is used to place the items, on average the number of remote accesses does not change. Thus, for all workloads the baseline system exhibits a similar (sub-optimal) behaviour. In the system running AUTOPLACER, locality is leveraged by relocating data items. As times passes, and more rounds of optimization take place, the system throughput increases up to a point where no further optimization is performed. It is interesting to note that, in case there is no locality, the throughput is not affected by the background optimization process. On the other hand, when locality exists, the throughput of the system optimized with AUTOPLACER is much higher than that of the baseline: it can be up to 6 times better for a workload



(a) Throughput with varying degrees of locality.



(b) % of remote operations.

Figure 3: AUTOPLACER performance

with 90% locality, and up to 30 times better in the ideal case of 100% locality.

Figure 3(b) helps to understand the improvement in performance by looking at the number of remote invocations that are performed in the system as time evolves. Since the initial setup relies on consistent hashing, both in the baseline and in the optimized system, the average probability of an operation being local is $\frac{1}{40} = 2.5\%$ for all workloads. Thus, when the system starts most operations are remote. However, the plots clearly show that the number of remote operations decreases in time when using AUTOPLACER. The plots also show another interesting aspect: although the number of remote operations decreases sharply after a few rounds of optimization, the overall throughput takes longer to improve. This is due to the fact that read transactions access a large number of objects, thus multiple rounds of optimization are required to alleviate the network, which is the bottleneck in these settings. This clearly highlights the relevance of the continuous optimization process implemented by AUTOPLACER. At the end of the experiment, the percentage of operations performed locally is already close to the percentage of locality in the workload; this shows that when the system stabilizes, AUTOPLACER was able to move practically all keys subject to locality.

Finally, Figure 4 compares the performance of AUTOPLACER and of a directory service-based system. These results were obtained by storing the data relocation map obtained at the end of the entire optimization process into a dedicated directory service. In this case, whenever a node requests a data item that is not stored locally, it contacts the directory service to determine its location, instead of querying the local PAA. The results clearly show that the additional latency for contacting the directory service can hinder perform significantly, independently of the locality of the workload. The plots highlight that, unlike for AUTOPLACER, the performance of directory-based systems can be worse than that achievable by using random placement. This is explainable considering that, with low locality, a large fraction of data accesses is remote, and that directory-based services impose a 2-hop latency, unlike consistent hashing and AUTOPLACER.

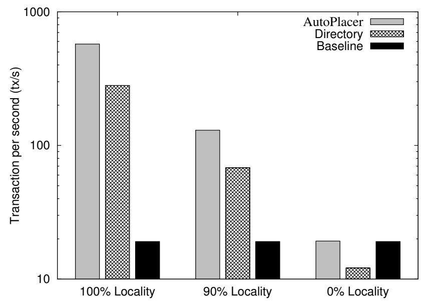


Figure 4: Throughput of AUTOPLACER, a directory-based and a consistent hashing-based solution, after a complete optimization process.

The speed-ups of AUTOPLACER vs the directory-based solution are significant, i.e. around 2x, even for the high locality scenarios. In these scenarios, the reduction of the number of remote operations leads to less lookups on the directory. However, the cost of accessing a remote data item is, in our testbed, about 2 orders of magnitude larger than that of accessing a local item. As a consequence, also in these scenarios, the cost of remote data accesses dominate the execution time of the requests. Hence, such requests, which suffer from one additional communication hop latency in a directory-based solution, effectively limit the throughput of such a solution leading to considerably worse results than AUTOPLACER.

7 Related Work

A common approach to implementing data placement mechanisms in large scale systems is to manage the data through coarse grain by partitioning it into buckets (also named directories [8] or tablets [6]). Through such partitioning, systems can deploy a centralized component which manages the location of all buckets in the system, moving them as required to balance the load on hotspot nodes. While coarse partitioning allows for somewhat manageable directories (maintaining the mapping between objects and nodes), on the other hand, it can reduce the effectiveness of the load balancing mechanisms. Furthermore, to improve data locality, these systems make use of sorted keys: the programmer is responsible for assigning similar keys to related data in order for it to be placed in the same server (or in the same group of servers) [8, 6, 18]. AUTOPLACER does not require the programmer to manually bucketize items. While we benefit from information enabled in the key structure, this information is not used for object placement, it is only used for optimizing the PAA. Also our system can establish a fine grain placement for the most accessed items.

As hinted several times in the paper, there is extensive work on defining optimal data placement strategies in multiple contexts. Many of these systems, such as Ursa [30] or Schism [9], attempt to perform optimization at a finer grain than buckets, but require the use of centralized components to compute the placement and to keep the resulting directories with the relocation map. As a result, they suffer from scalability limitations as the number of data items grows.

Several works have also attempted to derive distributed versions of the placement algorithm, to avoid the bottleneck of a single centralized component. The work by Leff *et al* proposes several distributed algorithms to approach the replica placement problem [20], and improvements to this work have been recently proposed in [19] and [31]. These results are not applicable to our system, as they only consider the placement of read-only replicas and not of the object ownership. Furthermore, this solution attempts to relocate all the data in the system, which may lead to scalability limitations similar to those of Ursa or Schism.

The work by Vilaça *et al.* [29] presents a Space-Filling Curves-based approach to placing co-related data in the same nodes by relying on user-defined per-object tags. The resulting system can provide good locality if the application is designed to perform actions using the tags, since nodes can locally determine who are the owners of the objects with specific tags. However, unlike our system, this placement is static and encoded by the programmer, and has no relation with the actual data access patterns that may emerge at runtime.

8 Conclusions and Future Work

This paper presented AUTOPLACER, a system aimed at self-tuning data placement in a distributed key value store. AUTOPLACER operates in rounds, and, in each round, it optimizes the placement of the top-k "hotspots", i.e. the objects generating most remote operations, for each node of the system. Despite supporting fine-grain placement of data items, AUTOPLACER guarantees one hop routing latency using a novel probabilistic data structure, the PAA, which minimizes the cost of maintaining and disseminating the data relocation map. AUTOPLACER has been integrated in a popular open source (transactional) key-value store, Infinispan, and experimentally evaluated using a porting of the TPC-C benchmark. The results shown that AUTOPLACER can achieve a throughput up to 6 times better than the original Infinispan implementation based on consistent hashing.

In this paper we have described how AUTOPLACER can be employed to optimize data placement in presence of static workloads. A detailed discussion and evaluation on how to extend AUTOPLACER to cope with variable workloads will be the subject of a future paper, but, below, we briefly describe a possible approach to achieve this result. AUTOPLACER can be made to operate in *epochs*. In each epoch, the system operates exactly as described in this paper. A new epoch is started when the need for recomputing data placement is identified, for instance, whenever an abrupt change of the remote access probability is detected [17] in the current epoch. As described in this paper, a new epoch e starts with an empty local lookup table $LookupT^e$ and, therefore, in the first iteration, all objects are considered when identifying hotspots. If objects need to be relocated (with regard to the previous epoch), their new position is stored in $LookupT^e$. In fact, the system described before can be seen as a particular case of the general algorithm, where only two epochs are considered: epoch o (defined by consistent hashing) and epoch 1 (the first workload).

Also the lookup function would have to be changed. Instead of consulting just the last lookup table, the lookup function would need to consult all the lookup tables in reverse chronological order. Naturally, this would slow down the lookup function after a long series of epochs. However, this could be easily mitigated by a background procedure that would merge the last lookup tables in a new consolidated table (in a process analogous to the one used in several log based file systems).

Acknowledgements

The authors wish to thank Zhongmiao Li for his valuable help with validating the initial prototypes of this work. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under projects PEst-OE/ EEI/ LA0021/ 2011, and CMU-PT/ELE/0030/2009 and by the Cloud-TM project (co-financed by the European Commission through the contract no. 257784).

References

[1] ALMEIDA, P., BAQUERO, C., PREGUIÇA, N., AND HUTCHISON, D. Scalable bloom filters. *Inf. Process. Lett.* (Mar. 2007).

[2] AMZA, C., COX, A., AND ZWAENEPOEL, W. Conflict-aware scheduling for dynamic content applications. In *Proc. of the 4th USITS* (Seattle (WA), USA, 2003).

[3] BLOOM, B. Space/ time trade-offs in hash coding with allowable errors. *Comm. of the ACM* (1970).

[4] CHANG, F., ET AL. Bigtable: a distributed storage system for structured data. In *Proc. of the 7th OSDI* (Seattle, USA, 2006).

[5] CHAZELLE, B., KILIAN, J., RUBINFELD, R., AND TAL, A. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. of the 15th SODA* (New Orleans (LA), USA, 2004).

[6] COOPER, B., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. In *Proc. of the 34th VLDB* (Auckland, New Zealand, Aug. 2008).

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st SoCC* (New York, NY, USA, 2010).

[8] CORBETT, J., ET AL. Spanner: Google's globally-distributed database. In *Proc. of the 10th OSDI* (Hollywood, CA, USA, 2012).

[9] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. In *Proc. of the 36th VLDB* (Singapore, Sept. 2010).

[10] DECANDIA, G., ET AL. Dynamo: amazon's highly available key-value store. In *Proc. of the 21st SOSP* (Stevenson, USA, 2007).

[11] DOMINGOS, P., AND HULTEN, G. Mining high-speed data streams. In *Proc. of the 6th KDD* (Boston, MA, USA, 2000).

[12] DOWDY, L., AND FOSTER, D. Comparative models of the file assignment problem. *ACM Computing Surveys* (1982).

[13] FLEISCH, B., AND POPEK, G. Mirage: a coherent distributed shared memory design. *SIGOPS Oper. Syst. Rev.* (Nov. 1989).

[14] GARBATOV, S., AND CACHOPO, J. Data access pattern analysis and prediction for object-oriented applications. *INFOCOMP Journal of Computer Science* (December 2011).

[15] JIMÉNEZ-PERIS, R., PATIÑO MARTÍNEZ, M., AND ALONSO, G. Non-intrusive, parallel recovery of replicated data. In *Proc. of the 21st IEEE SRDS* (Washington, DC, USA, 2002).

[16] KRISHNAN, P., RAZ, D., AND SHAVITT, Y. The cache location problem. *IEEE/ACM Transactions on Networking* (Oct. 2000).

[17] L., S., AND L., T. Cusum test for parameter change based on the maximum likelihood estimator. *Sequential Analysis: Design Methods and Applications* (2004).

[18] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* (Apr. 2010).

[19] LAOUTARIS, N., TELELIS, O., ZISSIMOPOULOS, V., AND STAVRAKAKIS, I. Distributed selfish replication. *IEEE TPDS* (Dec. 2006).

[20] LEFF, A., WOLF, J., AND YU, P. Replication algorithms in a remote caching architecture. *IEEE TPDS* (Nov. 1993).

[21] LEUTENEGGER, S., AND DIAS, D. A modeling study of the tpc-c benchmark. In *Proc. of the SIGMOD Conf.* (Washington, D.C., United States, 1993).

[22] MARCHIONI, F., AND SURTANI, M. *Infinispan Data Grid Platform.* PACKT Publishing, 2012.

[23] METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th ICDT* (Edinburgh,Scotland, 2005).

[24] MITCHELL, T. *Machine Learning*. McGraw-Hill, New York, 1997.

[25] PELUSO, S., ROMANO, P., AND QUAGLIA, F. Score: A scalable one-copy serializable partial replication protocol. In *Proc. of the 13th Middleware* (2012), pp. 456–475.

[26] PELUSO, S., RUIVO, P., ROMANO, P., QUAGLIA, F., AND RODRIGUES, L. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proc. of the 32nd ICDCS* (2012), pp. 455–465.

[27] RUIVO, P., COUCEIRO, M., ROMANO, P., AND RODRIGUES, L. Exploiting total order multicast in weakly consistent transactional caches. In *Proc. of the the 17th PRDC* (Pasadena, California, USA, Dec. 2011).

[28] STANOI, I., AGRAWAL, D., AND ABBADI, A. E. Using broadcast primitives in replicated databases. In *Proc. of the The 18th ICDCS* (Washington, DC, USA, 1998).

[29] VILAÇA, R., OLIVEIRA, R., AND PEREIRA, J. A correlation-aware data placement strategy for key-value stores. In *Proc. of the 11th DAIS* (Reykjavik, 2011).

[30] YOU, G.-W., HWANG, S.-W., AND JAIN, N. Scalable load balancing in cluster storage systems. In *Proc. of the 12th Middleware* (Lisbon, Portugal, 2011).

[31] ZAMAN, S., AND GROSU, D. A distributed algorithm for the replica placement problem. *IEEE TPDS* (Sept. 2011).

Adaptive Information Passing For Early State Pruning in MapReduce Data Processing Workflows

Seokyong Hong, Padmashree Ravindra, and Kemafor Anyanwu
Department of Computer Science, North Carolina State University
{shong3, pravind2, kogan}@ncsu.edu

ABSTRACT

MapReduce data processing workflows often consist of multiple cycles where each cycle hosts the execution of some data processing operators e.g., join, defined in a program. A common situation is that many data items that are propagated along in a workflow, end up being "fruitless" i.e. they do not contribute to the final output. Given that the dominant costs associated with MapReduce processing (I/O, sorting and network transfer) are very sensitive to the size of intermediate states, such fruitless data items contribute unnecessarily to workflow costs. Consequently, it may be possible to improve the performance of MapReduce data processing workflows by eliminating fruitless data items as early as possible. Achieving this will require maintaining extra information about the state (output) of each operator, and then *passing* this information to descendant operators in the workflow. The descendant operators can use this state information to prune fruitless data items from their other inputs. However, this process is not without any overhead and in some cases, its costs may outweigh its benefits. Consequently, a technique for adaptively selecting *Information Passing* as part of an execution plan is needed. This adaptivity will need to be determined by a cost model that accounts for MapReduce's partitioned execution model as well as its restricted model of communication between operators. These nuances of MapReduce impose limitations on the applicability of information passing techniques developed for traditional database systems.

In this paper, we propose an approach for implementing *Adaptive Information Passing* for MapReduce platforms. Our proposal includes a benefit estimation model, and an approach for collecting data statistics needed for benefit estimation, which piggybacks on operator execution. Our approach has been integrated into Apache Hive and a comprehensive empirical evaluation is presented.

1. INTRODUCTION

A dominant trend for large scale data intensive processing is to use parallel processing over a cluster of commodity grade machines. The MapReduce [13] parallel processing model that was made popular a few years ago by Google has emerged as the de facto standard for processing data-intensive workloads. Data intensive tasks are captured in the MapReduce model as workflows made up of sequences of MapReduce cycles/jobs. Each MapReduce cycle consists of 2 phases - a *Map* and a *Reduce* phase. The Map phase executes the `map` function which takes a set of key-value pairs as input, and maps each pair to an intermediate key-value

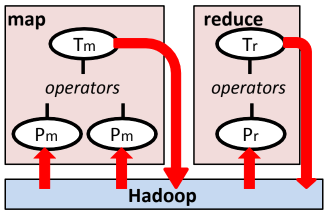


Figure 1: An Abstract MapReduce Job

pair. Each phase can have multiple instances (mappers and reducers respectively) running concurrently on assigned data partitions i.e., partition parallelism. A popular open-source implementation of Google's MapReduce proposal is Apache Hadoop [1]. In order to implement a data processing task in this model, programmers have to figure out the best translation of their tasks into the MapReduce model. This process has been simplified with the introduction of extended MapReduce platforms such as Hive [22] and Pig [21], that provide high-level declarative languages with querying constructs ala SQL, and compilers for automatically compiling high-level programs into MapReduce execution workflows. The compilation process assigns query operators or constructs to specific MapReduce cycles. Fig.1 shows an abstract data processing MapReduce model which captures the structure of what each cycle with data processing operators looks like using Hive as an example. *Primary operators* (P_m, P_r) are operators that take input data from the Hadoop framework, process them, and feed them into other operators. Once the operators in each phase complete their processing, a *terminal operator* (T_m, T_r) collects the resulting output. Further, for non-trivial data processing tasks that require multiple MapReduce cycles, control and data dependencies are implied by the high-level program and represented as a workflow.

An important thing to note about MapReduce data processing workflows is that they can be very costly. Table 1 shows the dominant costs (CPU costs of map and reduce function are ignored) in a MapReduce cycle. The scheduler schedules a set of slave nodes (*mappers*) to execute the Map phase, and assigns a split or partition of the input file to each mapper. The input data loading cost is represented as C_{Load} in Table 1. (Note that multiple splits can be assigned to a mapper, in which case multiple `map` function instances are executed on the node). Once all mappers are complete, the intermediate key-value pairs are materialized on the mappers' local disk (cost $C_{MapStore}$) in preparation for the Reduce phase. The scheduler then assigns nodes (*reducers*) to *reduce* a partition of map output values in a

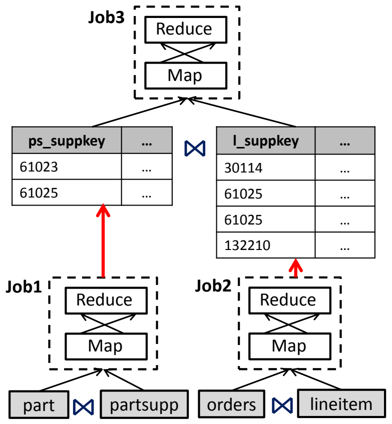


Figure 2: Unnecessary data movement of "fruitless" data items across a workflow

process that consists of three phases: *copy* — copying the sorted map output from mappers' disks to reducer nodes, resulting in a data transfer cost $C_{Shuffle}$; *merge* — merging the sorted output lists from different mappers based on the intermediate key with cost C_{Merge}, and *reduce* — `reduce()` is invoked once for each intermediate key, and applied to the associated group of values. The output key-value pairs generated by all reducer instances are merged and materialized into the underlying distributed file system e.g., Hadoop Distributed File System (HDFS) contributing to cost $C_{RedStore}$. The cost of a workflow is the aggregate cost of all cycles in the workflow. As can be observed, these costs are all sensitive to the size of data. Therefore, developing ways to keep the footprints of intermediate states small is crucial to performance of MapReduce workflows.

Table 1: Cost Factors in a MapReduce Job

Cost Factor	Description
C_{Load}	Input data loading cost
C_{Sort}	Map-side sorting cost
$C_{Shuffle}$	Data transfer cost
C_{Merge}	Reduce-side partition merging cost
$C_{MapStore}$	Map output data materialization cost
$C_{RedStore}$	Reduce output data materialization cost

1.1 The Problem

Due to the significant overhead associated with each MapReduce cycle, a key objective when optimizing MapReduce data processing workflows is minimizing the length of the workflow i.e., the number of MapReduce cycles in the workflow. Another very important objective (similar to relational databases) is minimizing the overall size of intermediate results or states produced during a data processing workflow (query). This is particularly crucial because of the multiple I/O, sorting, and network data transfer phases of intermediate results during a MapReduce data processing workflow. State-producing operators such as the `JOIN` operator, whose outputs may be larger than their inputs are an important consideration when thinking about minimizing the size of intermediate states produced. Indeed, in relational query optimization, the `JOIN` operator is a very fundamental operator for combining datasets, and its optimization is the focus of a lot of research. Typically, this is achieved by ordering operators in a way that minimizes inputs to each join. Cost models based on heuristics are used to estimate outputs of each operation so as to determine the best ordering of operations. In order to achieve this during compile-time, the parameters to such cost models will have to be precomputed, requiring preprocessing of data. This is natural for relational databases where data structures like indexes and statistical profiles of data are maintained as data is ingested into the system. However, MapReduce data processing platforms are typically not used to manage or host data in the long term but rather just for processing data-intensive workloads. Thus, features like statistics and indexing are immature or absent in MapReduce-based platforms like Hive and Pig. Further, any cost models used for relational optimization are not adequate for MapReduce data processing, because they do not capture key MapReduce-specific costs. Consequently, it is very important to consider runtime optimization techniques that can be applied as data is being ingested. Further, since these platforms are often used in batch processing mode, information gathered during earlier tasks in the batch workload can be used to inform the execution of latter tasks within the same batch workload.

A useful group of runtime optimization techniques in relational query optimization is called *Information Passing*(IP). Such techniques support passing information from earlier phases of data processing to later execution steps, particularly join operations, to help them prune their states more aggressively. If this is done early enough, we could avoid fruitless data from traveling along in the workflow only to be eventually pruned. As an example, Fig.2 shows an example job plan consisting of three MapReduce jobs executing equi-join (joining relations on equality condition on particular fields). The execution sequence is $Job1$, $Job2$, and $Job3$ (all joins are repartitioning joins). $Job3$ joins the two intermediate tables on *ps_suppkey* and *l_suppkey* after $Job1$ and $Job2$ complete. In this case, it is obvious that only records with suppkey 61025 will be joined and emitted to the final output (output of $Job3$) while remaining records will be discarded. The other records are essentially fruitless or irrelevant to final result. Unfortunately, these fruitless records affect C_{Load}, C_{Sort}, $C_{MapSort}$, $C_{Shuffle}$, and C_{Merge} costs for $Job3$. Further, they contribute to the costs of $Job1$ and $Job2$. If on the other hand, it is possible to pass information about the output context of $Job1$ to $Job3$, then it is possible to have $Job3$ prune its inputs (output of $Job2$ and $Job1$) while loading them, so that some savings in its C_{Sort}, $C_{MapStore}$, $C_{Shuffle}$, and C_{Merge} can be achieved. Better still, we may be able to pass this information to $Job2$ so that while it writes its output, it can prune records that are irrelevant to final results e.g., records 30114, 132210. In addition to savings in $Job3$'s costs, we can get additional savings with respect to $Job2$'s $C_{RedStore}$ and $Job3$'s C_{Load}. Such savings can be significant if pruning irrelevant tuples can be done much earlier in the workflow. For example, assume an execution plan such that these records are eventually pruned in $Jobk$ and not in $Job3$. Then, pruning these records from Job2's output will avoid carrying them along and processing them in some of jobs in $Job3$ to $Jobk - 1$, before being eventually pruned out in $Jobk$.

In shared-memory or distributed environments where relational information passing techniques have been investigated, IP is usually achieved through some centralized shared memory structure which can be accessed by all operators. Shared nothing cluster environments and rigid communication models allowed by MapReduce make such an implementation strategy difficult. These issues will be elaborated in

Section 2.1. More significant is that the overhead of information passing may surpass its benefits. Therefore, techniques for runtime, adaptive selection of an information passing strategy as part of workflow execution need to be developed. Further, the decision for information passing selection has to be based on a cost model that is MapReduce-aware and informed by characteristics of data as determined during processing. The latter also suggests the need for a light-weight technique for collecting information about data at runtime by piggybacking on data processing.

1.2 Contributions

Specifically, we make the following contributions:

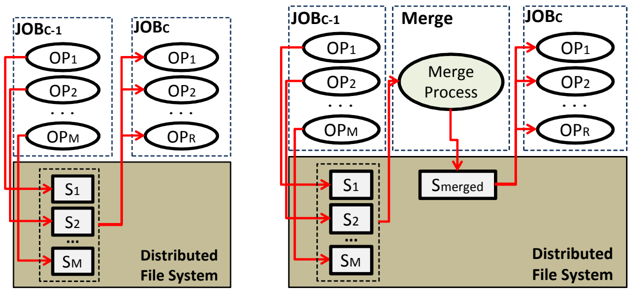
- We propose an architecture that supports adaptively enabling information passing (IP) in Hadoop-based data processing platforms based on its cost-benefits tradeoffs. The architecture provides support for *IP-aware* operators, IP-plan compilation and execution. We present a strategy for integrating these components into Apache Hive.
- We propose a MapReduce-based *benefit estimation* cost model for estimating the benefits of an IP-enabled execution plan. In addition, we present a light-weight data statistics collection technique that piggybacks on workflow execution, and collects necessary parameter values for the IP benefit estimation cost model.
- Finally, we present a comprehensive evaluation of the proposed framework using two datasets including a benchmark.

2. RELATED WORK

2.1 Sideways Information Passing

Relational database research has proposed a few variants of *Sideways Information Passing* (*SIP*) techniques [19, 17, 8, 9, 20] where information is passed between operators in an execution plan. *Magic-set rewriting* [19] ships summary information on examined values from a parent query to its subqueries or views so that they can discard unmatched records. *Semi-join* [9] processes join over remote sites. It projects and sends join columns from one site to another, and joins the projection with a remote relation. The resulting records are transferred to the original site and joined. In a sense, the task of passing information is integrated into the semantics of the operator itself rather than being an augmentation. In [8], an operator named *Eddy* routes records among relational operators based on dynamic runtime properties so as to maximize performance. [17, 20] produce filters from operators to other co-related operators to prune unnecessary records. Compile-time plans are augmented by run-time benefit estimation to reduce the information passing overhead. [17, 8, 20] introduced adaptive SIP approaches. Adaptivity can selectively add IP to an execution plan based on a cost model that relies on statistics of the underlying database system.

Discussion. While these SIP techniques have similar goals with our proposal, their implementations make assumptions that do not carry over to MapReduce execution platforms, making their adoption infeasible. Specifically, [19, 20] is designed for a centralized shared-memory environment where summaries can be exchanged using buffers or other message passing techniques [17, 8] that assume the existence of a centralized repository for exchanging tuples or summaries at will. On the other hand, MapReduce executes in a shared-nothing environment with a very restricted communication model between nodes. Further, operator execution in existing techniques is holistic i.e, one instance of operator processes all the input for that operator. This is in contrast to partitioned execution in MapReduce where multiple instances of an operation are executed, each processing one partition of the operator's input. The consequence is that information about the state of a MapReduce operator is fragmented across the different instances whereas in the traditional case, all summary information can be found in a single location. Finally, the adaptivity model proposed in [17] assumes the availability of data statistics. However, as was mentioned before, such features are lacking in existing MapReduce data processing platforms or require significant overhead [15] to compute. These assumptions by existing approaches make for a much simpler information passing problem than would be required in the MapReduce setting.



(a) (b)

Figure 3: Summary Distribution

2.2 Information Passing Other Join Optimizations for MapReduce Platforms

Some recent efforts [10, 16] have focused on enabling information passing in MapReduce. A series of semi-join techniques in [10] resemble the traditional 2-way semi-join. However, each join operation is implemented using three MapReduce cycles which can be very expensive. In some earlier work [16], we proposed a basic information passing technique *Hadoop Information Passing (HIP)* that enables summary exchanges between multiple MapReduce jobs in a workflow. In this approach, fragmented summaries about the state of an operator are generated at the end of the cycle in which its executes. The summaries are stored into the Hadoop Distributed File System as compressed files of summary lists. If the total size of those summary fragments do not exceed a user-defined threshold, they are broadcast to nodes at the initiation of a subsequent job that takes as input, the earlier job's output. Other relevant work include efforts to reduce the length of a MapReduce workflow by clustering operations into few cycles as possible using multi-way join algorithms [18, 7, 23]. If the length of a workflow is reduced to just one cycle, then information passing becomes unnecessary. However, this is not always feasible for non-trivial data processing tasks because invariably, different operators will have conflicting key partitioning requirements forcing their execution to be assigned to different cycles. Also, some of the algorithms result in a large amount of replication which impedes performance. In the case of [23], a cost-based query optimization approach is proposed, where the multiplicity of

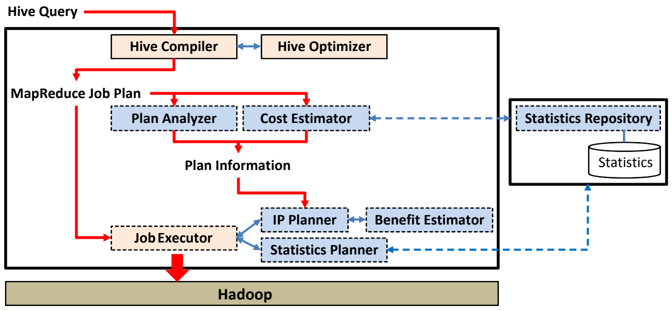


Figure 4: System Architecture (Solid-line: Hive component, Dotted-line: New or modified component)

replicated-join is decided by cost estimation. However, this approach requires additional MapReduce cycles per input table to calculate the necessary statistics.

3. ADAPTIVE INFORMATION PASSING IN HADOOP WORKFLOWS

In this section, we discuss our proposal for enabling information passing on Hadoop-based processing systems. Fig.4 shows our extended Hive framework with dotted lines denoting the new or extended components. Our system consists of three major components, (i) an *information passing framework* that enables IP-aware plans to generate and utilize summaries across jobs, (ii) a *benefit estimator* that estimates the costs for summary generation and propagation for each job, and (iii) a *statistics collector* that piggybacks on the MapReduce execution process to collect statistics required by the benefit estimator.

3.1 Information Passing Framework

At compile-time, this framework decides whether information passing or statistics generation should be enabled by contacting several components. The default Hive job executor was modified to enable and execute IP-related decisions.

Compile-time preparation. A *Plan analyzer* analyzes a MapReduce job plan to produce *plan information*. Plan information contains job-specific information and relationships among the multiple jobs in a job plan. Information for each job is stored in a data structure called *JobDescriptor*, which maintains the job ID, a pointer to the Hive job descriptor, the type of reduce-side primitive operator P_R (e.g., join or group-by), paths for summary input and output, and so on. The relationships among jobs are represented in a *dataflow graph* and a *dependency graph*. A dataflow graph describes the input/output dataflow of jobs, and a dependency graph (reverse of the dataflow graph) describes their execution order. A *job executor* that submits each job in a MapReduce job plan to the Hadoop framework, references the plan information to make a decision on information passing. A *cost estimator* retrieves statistics about the job's input tables from the *statistics repository*, and calculates job costs. Job descriptors are augmented with such job cost information. Before submitting the current job J_C, the job executor invokes the *IP planner* to check whether the job should generate summary information ($SUMMARY\_CREATION$) for any subsequent jobs, and/or load any summaries created by previous jobs ($SUMMARY\_USAGE$).

Algorithm 1 corresponds to the decision-making process for summary generation, which consists of two steps. First, the IP planner probes the plan information and checks whether any subsequent job processing a join operation can leverage the summary information that J_C may produce (lines 3-4). For example, job J_{N-1} in Fig.5a is eligible for summary generation since job J_N can use the summary to prune unnecessary data while loading $TableA$. Next, the IP planner calls the *benefit estimator* to estimate the benefit that the summary can bring about (line 5). If there is considerable benefit (configurable via parameter $\beta > 0$), the planner makes a decision to enable $SUMMARY\_CREATION$ (line 6) so that J_C generates summary information during its execution.

Algorithm 1: Decision-making for State Creation

```
1 $SUMMARY\_CREATION \leftarrow false$;
2 if $IP$ is enabled in configuration file then
3     if $J_C$ linked to any job $J_N$ in dataflow graph then
4         if $descriptor(J_N).ReducerType == JOIN$ then
5             if $BenefitEstimator.estimate() > \beta$ then
6                 $SUMMARY\_CREATION \leftarrow true$;
              end if
          end if
      end if
  end if
```

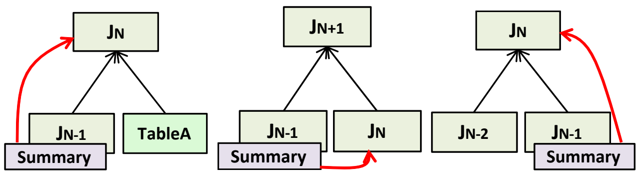


Figure 5: Possible IP Plans (a) Map-side pruning using *child-to-parent* IP, (b) Reduce-side pruning using *sibling-to-sibling* IP, (c) another example of *child-to-parent* IP using intermediate tables

The IP planner also examines J_C to determine usability of available summaries from a previously executed job J_P. Algorithm 2 shows the corresponding decision-making algorithm for summary utilization which considers two possible cases of information passing: *child-to-parent* and *sibling-to-sibling*. Fig.5 shows three job plans illustrating the two scenarios (the execution sequence is J_{N-2}, J_{N-1}, J_N, J_{N+1}, and current job $J_C = J_N$). First is the case of *child-to-parent* IP between a parent job J_C and a child job J_P (lines 3-7). If job J_C has a join operator in its reduce-phase (line 3), the planner looks up the dependency graph to find any jobs whose output is fed into J_C i.e., child jobs which have generated summaries that have not yet been used (lines 4-5). There are two possible cases of child-to-parent IP: a job may either join an intermediate table and a base table (Fig.5a), or it may join two intermediate tables (Fig.5c) such that the summary generated by the child was not used by its sibling. In both cases, the planner determines whether the size of the summary files P_i ($1 \leq i \leq r$, for r reducers) generated by the child job is less than a user-defined threshold α. This is done to avoid heap memory leakage issues caused by loading large summary information. If the summary size is less than the threshold, the planner invokes the benefit estimator to determine the benefit in summary usage. Based on the estimated benefit, the summary from the child job is used by J_C to prevent irrelevant data from being shuffled between map and reduce phases. This map-side pruning helps to reduce C_{Sort}, $C_{Shuffle}$, and C_{Merge} costs in J_C. The second case is that of *sibling-to-sibling* IP where the summary information generated by a previously-executed sibling job J_S in the dataflow graph, can be used to prune the output of J_C (line 8). For example, J_N in Fig.5b has a sibling job (J_{N-1}) which generated a summary. J_N uses the summary to curtail C_{Store} costs in J_N, and C_{Load}, C_{Sort}, $C_{Shuffle}$, and C_{Merge} in J_{N+1}. Before the job executor submits each job to Hadoop, all decisions are encoded in the job so that corresponding operators generate and / or load summaries.

Algorithm 2: Decision-making for State Utilization

```
1  $SUMMARY\_USAGE \leftarrow$ false ;
2  if $IP$ is enabled in configuration file then
       //Child-to-Parent IP
3      if $descriptor(J_C).ReducerType == JOIN$ then
4          if $J_C$ has a neighbor $J_P$ in dependency graph then
5              if $J_P$ generated summaries $P_i$ $(1 \leq i \leq r)$ such that $\sum |P_i| \leq \alpha$ then
6                  if $BenefitEstimator.estimate() > \beta$ then
7                      $SUMMARY\_USAGE \leftarrow$ true;
           //Sibling-to-Sibling IP
8          if $J_C$ has a sibling job $J_S$ in dataflow graph then
9              if $J_S$ generated summaries $P_i$ $(1 \leq i \leq r)$ such that $\sum |P_i| \leq \alpha$ then
10                 $SUMMARY\_USAGE \leftarrow$ true;
```

Algorithm 3: IP-Aware map()/reduce() Skeletons

```
   //Reduce() of current job $J_n$
   Reduce (key:grpKey, val:Corresponding list of tuples T)
   Init ();
   reduce ()
1  foreach $tu \in T$ do
2      $out\_tup \leftarrow$ Normal reduce processing;
3      $next\_join\_key \leftarrow$ extract join key for next job;
4      $hashVal \leftarrow next\_join\_key$.hash();
5      Add distinct $hashVal$ to in-memory sorted set;
6      emit <null, $out\_tup$>;
   Close ()
7  $status \leftarrow$ close status of all reduce operators;
8  if $status$ is Success then
9      $summary \leftarrow$ bloom filter on $hashVals$;
10     Store $summary$ to HDFS;

   //Merge summary from $r$ reducers
11 $mergedSummary \leftarrow$ merge($summary_1$,...,$summary_r$);
   //Map() of a subsequent job $J_{n+1}$
   Map (key:null, val:Tuple tup from Input)
   Init ()
12 Load $mergedSummary$;
13 $summarySet \leftarrow$ build in-memory sorted set from $mergedSummary$;
   map ()
14 $join\_key \leftarrow$ extract the join column from $tup$;
15 if $tup \in baseRelation$ then
16     if $join\_key.hash()$ does not exist in $summarySet$ then
17         Prune out $tup$;
18 $out\_tup \leftarrow$ Normal map processing;
19 emit <$join\_key$, $out\_tup$> ;
   Close ();
```

Run-time operation. In order to generate summary information and utilize it, operators in a job should be aware of decisions and operate accordingly. Hence, we designed *IP-aware* reduce-side T_r and map-side P_m operators. As described earlier, T_r is the final logical operator in the reduce phase that stores the reduce output to HDFS. If a job should generate summary information ($SUMMARY\_CREATION$ is true), decision-aware T_r operator generates bloom filters on the target column that is a join key in a subsequent job. Algorithm 3 describes the pseudo code for the IP-Aware *map* and *reduce* function skeletons. Note that the init (close) function calls each operator's init() (close()) method in the beginning (end) of each phase. At runtime, the IP-aware T_r operator calculates the hash value on the target column for each record, and puts the hash value in an in-memory buffer (lines 3-5). In the close() of the operator, the hash values are compacted using a bloom filter to minimize the size of the summary information (lines 8-10), and stored into the HDFS. Multiple instances of the T_r operator produce multiple partial summaries (one per reducer), and the job executor merges them into a single summary (merged bloom filter) as shown in Fig.3b.

The merged summary is broadcasted to computing nodes via *JobConf* before the subsequent job executes. JobConf is a data transport facility provided by the Hadoop framework to propagate system-wide and job-specific configurations to nodes. The JobConf is copied once to each node's local disk rather than to every mapper or reducer, and hence reduces the summary propagation costs. At the initialization phase of the subsequent job, the IP-aware map-side operator P_m loads the merged summaries into an in-memory buffer (lines 12-13). Whenever a record is read, the operator calculates the hash value for the target column (join key), probes the in-memory buffer, and prunes out irrelevant records (lines 15-17). Note that in the case of sibling-to-sibling IP, the summary loading and pruning happens in the reduce phase using the IP-aware T_r operator. If the T_r operator also needs to generate summary, the pruning precedes the summary generation phase. The next section describes the benefit estimation model that guides decisions in the information passing framework.

3.2 Benefit Estimation Model

Our benefit estimation model does not consider the case of sending summaries to sibling jobs (e.g., Fig.5b). Instead, it estimates the benefit from consuming summaries in parent jobs (e.g., Fig.5a and 5c). If a given summary is beneficial for a parent job, it is expected that a sibling job can achieve more benefit by using the summary in terms of C_{STORE} and C_{LOAD}. Hence, the proposed benefit estimation model aggregates partial benefits in C_{SORT}, $C_{SHUFFLE}$, and C_{MERGE} (B_{sort}, $B_{shuffle}$, and B_{merge} respectively), and differences the aggregated benefit and the cost for summary creation and propagation ($C_{summary}$):

$$B = B_{sort} + B_{shuffle} + B_{merge} - C_{summary} \tag{1}$$

Benefit in each step is estimated by subtracting the cost in the IP-enabled approach from the cost in the default approach. Hence, benefits in the three steps are:

$$B_{sort} = C_{sort-orig} - C_{sort-ip} \tag{2}$$

$$B_{shuffle} = C_{shuffle-orig} - C_{shuffle-ip} \tag{3}$$

$$B_{merge} = C_{merge-orig} - C_{merge-ip} \tag{4}$$

.

In sort-phase, MapReduce performs external merge sort and the cost can be estimated as follows:

$$C_{sort-orig} = M(C_R + C_W)(\lceil \log_{B-1} \frac{M}{B} \rceil + 1) \quad (5)$$

when each mapper emits intermediate data of M pages on average, and the size of sorting buffer is B. C_R (C_W) is the unit cost to read (write) a page from (to) disk. Let Δ be the average size of unnecessary data that is expected to be pruned by a given summary information, then the cost in the IP approach is:

$$C_{sort-ip} = (M - \Delta)(C_R + C_W)(\lceil \log_{B-1} \frac{M-\Delta}{B} \rceil + 1) \quad (6)$$

.

When the intermediate data is shuffled into r reducers, each reducer receives, on average, $(M \times m)/r$ pages from m mappers. Hence, the estimated data shuffle cost of the original approach is:

$$C_{shuffle-orig} = \frac{M \times m}{r} C_T \quad (7)$$

where C_T is the unit cost to transfer a page. The estimated data shuffle cost in the IP approach is:

$$C_{shuffle-ip} = \frac{(M-\Delta)m}{r} C_T \quad (8)$$

In reduce-phase, each reducer receives partitions from all m mappers and merges them into one block. Hence, we estimate the merge cost in the default approach as equation 9, and that of the IP approach as equation 10.

$$C_{merge-orig} = R(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (9)$$

$$C_{merge-ip} = (R - \Delta \cdot \frac{m}{r})(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (10)$$

where R is the average input size per reducer. Since $M \cdot m = R \cdot r$, we substitute $(M \cdot m)/r$ for R producing the following equations:

$$C_{merge-orig} = M \cdot \frac{m}{r}(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (11)$$

$$C_{merge-ip} = \frac{m}{r}(M - \Delta)(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (12)$$

Cost $C_{summary}$ is caused by summary generation and propagation and consists of four costs:

$$C_{summary} = C_{ip-store} + C_{ip-merge} + C_{ip-copy} + C_{ip-load} \quad (13)$$

Each T_r operator stores a partial summary information in a bloom filter of size $|F|$ into HDFS. Hence, the cost to store a partial summary is:

$$C_{ip-store} = (U_T + U_W)|F| \quad (14)$$

where U_R, U_W, and U_T are unit costs to read, write, and transfer a single byte, respectively. Since partial summaries generated by r' reducers are combined into one bloom filter, the cost to merge partial summaries is:

$$C_{ip-merge} = (U_T + U_R)|F|r' + (U_T + U_W)|F| \quad (15)$$

The cost to copy a merged summary to nodes is:

$$C_{ip-copy} = N(U_R + U_W + U_T)|F| \quad (16)$$

The summary loading cost of each operator from local disk is:

$$C_{ip-load} = U_R|F| \quad (17)$$

Parameters such as C_R, C_W, C_T, U_R, U_W, and U_T are machine-specific and can be earned by performance measurement. B and N can be calculated with Hadoop configuration parameters, and $|F|$ is set in the Hive configuration file. At the time that benefit estimation is performed, the number of reducers (r') in a current job has already been set by the Hive compiler. However, the number of mappers (m) and reducers (r) in its parent job are unknown since they depend on the input table sizes of the parent job and it is complicated to estimate those parameters without input statistics. For the same reason, the estimation of M and Δ is complicated. In the next section, we describe our statistics collection approach that piggybacks on MapReduce job execution.

3.3 Piggybacking Statistics Collection on Operator Execution

Our approach for statistics collection piggybacks on the execution of MapReduce jobs. Generated statistics are used by the cost estimator to estimate statistics on intermediate data in a MapReduce job plan. These statistics are used by the benefit estimator to calculate necessary parameters such as m, r, M, and Δ, which are supplied to the benefit estimation model to make decisions about information passing. Required statistics include the size of tables, the number of records, the average sizes of columns, and the value distributions within a column. The general idea is that rather than requiring a separate pre-processing step for computing statistics, we choose to exploit the fact there is a good likelihood that users will want to execute multiple queries on their datasets. Therefore, for each query, we compute and store statistics on selected columns of the tables being processed. Then, for future queries that refer those columns, we use the already computed statistics. Note that the statistics planner does not estimate statistics on all columns in input tables. For example, while evaluating TPC-H queries, columns for comments (e.g., *ps_comment*, *l_comment*, *c_comment*, and *p_comment*) and auxiliary information (e.g., *c_address*, *s_address*, and *s_phone*) are rarely positioned in filter or join conditions, and hence are not important for cost estimation. Such columns can be excluded to reduce the overhead of statistics generation and transmission costs. Users can manually specify additional columns into the column set by updating the Hive configuration file.

At compile-time, the *statistics planner* creates a list of input tables from a MapReduce job plan, and extracts a set of meaningful columns whose statistics are required for cost estimation. The statistics planner probes the statistics repository using Java Remote Method Invocation (Java RMI) to check the availability of statistics related to the listed tables and columns. If the statistics are unavailable, the statistics planner enables statistics collection and embeds the decision into the job plan. Additionally, the statistics planner registers the input table sizes in the statistics repository if such information is missing.

At run-time, the map-side primary operators (P_m) in each job generate the statistics on the required columns. Once the operators finish execution, the generated statistics are registered with the statistics repository. The generated statistics are partial since the P_m operators process records from different input portions. In this environment, it is straightforward to calculate the number of records and the average sizes of columns from fragmented statistics. However, reconstructing the value distributions from fragmented statistics is quite tricky. Sampling [11, 12] or histogram [23] can be considered but such approaches may incur high inaccuracy or huge data transmission overhead. In order to combat this issue, we used a recent logarithmic counting method called as *HyperLogLog* [14]. This algorithm supports set union operation which enables combination of multiple statistics fragments in a natural way. Therefore, partial value-distribution from multiple input partitions can be merged efficiently. Next, the memory footprint that the algorithm requires to store bit-vectors is relatively small ($O(\log\log D)$) with high accuracy. For example, the algorithm consumes 1.5KB memory with a 2% error rate for 10^9 cardinality values [14]. When a job completes its execution, the statistics repository unions partial bit-vectors that have been registered.

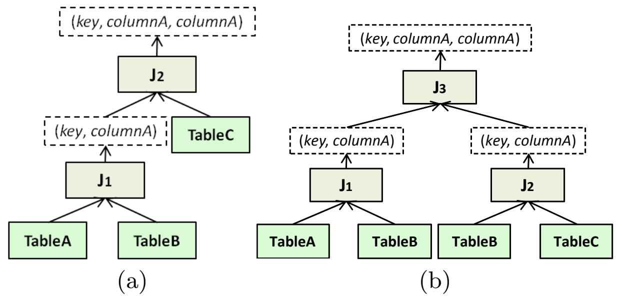


Figure 6: (a) Query1, (b) Query2

4. EXPERIMENTAL EVALUATION

The information passing approach has been implemented in Apache Hive 0.5 [2] on top of Apache Hadoop 0.20.0 [3]. This section presents an extensive study of the proposed approach by comparisons with original Hive, HIP [16], and semi-join [10]. For HIP, the threshold of total summary size was set to 512KB and 4MB. For the proposed IP approach, two sizes of bloom-filter were used for storing summary information (512KB and 4MB). First two steps of semi-join were implemented using vanilla MapReduce applications. However, the last step was implemented as a repartitioning join, instead of Hive's fragment-replication join named *map-side join*. This is because the map-side join failed for experiments in which the size of intermediate data from the first two steps was not small enough to fit into memory.

4.1 Experiment Setup

Experiments were conducted on a cluster on *NCSU VCL* [6] which consists of 21 blade servers (one master node and 20 slave nodes). Each node has a 3.0GHz dual-core Xeon processor, 4GB memory, and a 28GB SCSI disk, and runs Red Hat Enterprise Linux 5. Hadoop framework was configured with 512MB of block size, replication factor 1, no speculative execution, and 1024MB of heap size for mappers and reducers.

4.2 Workloads and Analysis

Two kinds of synthetic datasets, including one benchmark were used. The first synthetic dataset is generated based on user-supplied parameters such as number of records, column sizes, and range of join column values. This benchmark was chosen since it is complicated to manually set reference ratios among generated tables with existing benchmarks. Three tables were generated using this benchmark: *TableA*, *TableB*, and *TableC*. Each table is 20GB and includes three columns (*key*: 25B, *columnA*: 75B, *columnB*: 100B). The *key* column in each table was used as a join key. The join key density of *TableB* and *TableC* is fixed to one while the join key density in *TableA* varies across experiments. Hence, records in *TableA* match different numbers of records in other tables. Records in *TableB* and *TableC* are exactly same. In order to evaluate the effects of passing summary information to different MapReduce jobs, two benchmark queries were used as shown in Fig.6. The query in Fig.6(a) compiles into two MapReduce jobs. The first job joins *TableA* and *TableB*, producing intermediate records of (*TableB.key*, *TableA.columnA*). The second job joins the intermediate output from the first job with *TableC*, and generates (*TableB.key*, *TableA.columnA*, *TableC.columnA*). For HIP and IP, the execution time of *Job2* is summed up with any delay in execution time of *Job1*. Semi-join was computed on the intermediate records from *Job1* and *TableC*, and the execution times of three steps were summed up. Next, the query in Fig.6(b) is translated into three MapReduce jobs. The first one is similar to the first job in the previous query. The second job joins *TableB* and *TableC*, and produces same number of records. The intermediate records are of (*TableB.key* and *TableC.columnA*). The last job joins the two intermediate tables and generates records of the form (*TableB.key*, *TableA.columnA*, *TableC.columnA*). Execution times for *Job2* and *Job3* for different approaches were also compared. However, semi-join was not performed for this query since the first two steps of semi-join cannot remove any records deriving no benefit. Second set of experiments used the TPC-H benchmark [5] dataset (40GB). The Hive version of TPC-H queries [4], which are written in HiveQL were used. Among the TPC-H benchmark queries, a set of relevant queries with multiple join operations were chosen. Different query plans were evaluated.

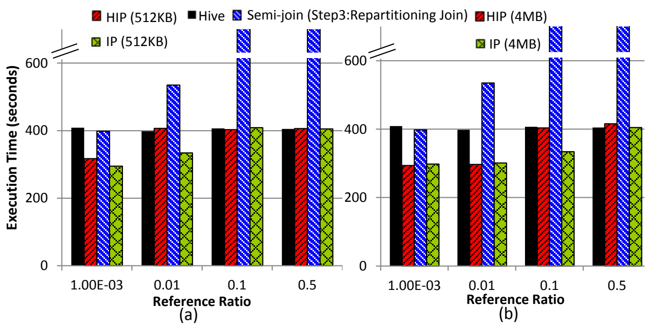


Figure 7: Execution times for Query1:Job2 with varying reference ratios (a) HIP: 512KB threshold, IP: 512KB bloom-filter, (b) HIP: 4MB threshold, IP: 4MB bloom-filter

4.3 Experimental Results

Performance improvement with varying reference ratios. The IP approach enhances query processing performance by pruning unnecessary records before being joins. Hence, the effectiveness of the approach depends on the reference ratio between joined tables. To check the relationship between performance and reference ratio, *TableA*'s reference ratio was varied such that its records match different numbers of records in the other tables. Fig.7 shows

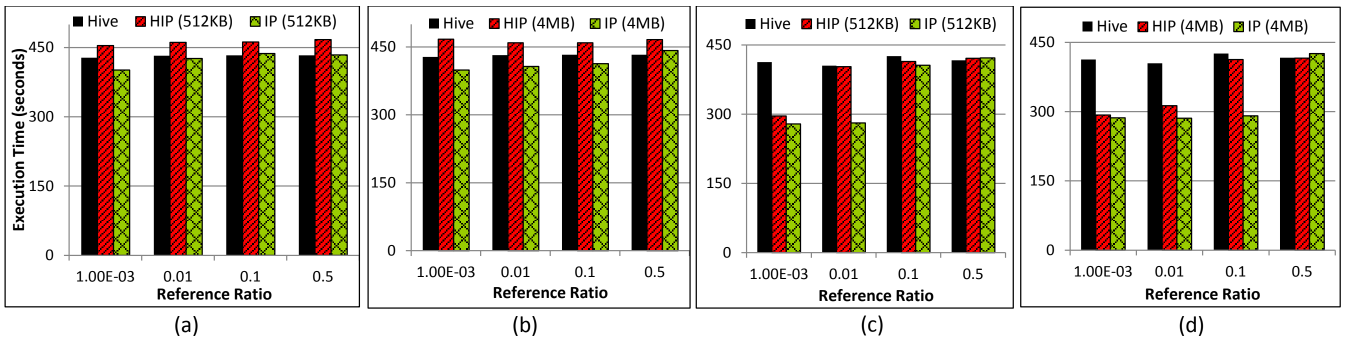


Figure 8: Query2 execution times with varying reference ratios, (a) Job2(HIP: 512KB threshold, IP: 512KB bloom-filter), (b) Job2(HIP: 4MB threshold, IP: 4MB bloom-filter), (c) Job3(HIP: 512KB threshold, IP: 512KB bloom-filter, (d) Job3(HIP: 4MB threshold, IP: 4MB bloom-filter))

the execution times of the query in Fig.6(a). First, semi-join failed for reference ratio 0.1 and 0.5 (denoted by broken lines on the y-axis to indicate that execution failed after a lengthy execution period) since the size of join key list exceeded available heap memory in its first step. For reference ratio 0.001 and 0.01, the semi-join approach did not show better performance than the original Hive approach. On the other hand, HIP and IP showed performance improvements as the reference ratio decreases. However, the IP approach was beneficial with relatively higher reference ratio than HIP (reference ratio = 0.01 in Fig.7(a) and reference ratio = 0.1 in Fig.7(b)). A problem with HIP approach is that it cannot calculate the size of summary information before partial summary fragments are merged and loaded into memory. Hence, the total size of summary fragments is compared with the user-defined threshold. As a consequence, even if the size of merged summary information is less than the threshold, it may disallow generated summary information to be loaded by jobs. On the other hand, IP approach decides about summary generation and utilization based on benefit estimation, and can maximize the benefit of information passing. In Fig.7(b), for example, HIP disabled the use of summary for reference ratio 0.1, while IP allowed summary to be used by *Job2* deriving benefit.

When HIP processes *Query2* in Fig.6(b), *Job1* and *Job2* generate summaries, and *Job3* utilizes those summaries. Hence, as shown in Fig.8(a-b), *Job2* execution times in HIP are longer than the original Hive approach due to summary generation overhead. On the other hand, the IP approach transports summary information from *Job1* to *Job2*, and prunes unnecessary records at the end of *Job2*. Hence, it could derive benefits in materialization steps when its benefit estimation enables summary information (e.g., when reference ratio = 0.001 and 0.01 in Fig.8(a) and reference ratio = 0.001, 0.01, and 0.1 in Fig.8(b)). The benefits were less than 8% performance improvement. In case of *Job3*, IP achieved more performance improvement than HIP since reduced intermediate data from *Job2* derived benefits in data loading phase in addition to shuffle-phase as shown in Fig.8(c-d).

Performance improvement with varying block sizes. This experiment evaluates the impact of varying HDFS block size on the different approaches. The reference ratio was fixed to 0.01, and HIP threshold and size of bloom-filter in IP were set to 4MB. Fig.9(a) shows the performance improvement rates of *Job2* in *Query1* relative to the original Hive approach. Semi-join was worse than Hive in all settings while the performance degradation rate decreased as block size increased since the execution time of semi-join increased slower than that of Hive. Both HIP and IP showed linear speedups with increasing block sizes. Each mapper that loads a block from *TableC* is assigned more data as the block size increases. Hence, given the summary information produced in the previous job, both approaches are able to prune out larger amount of “fruitless” data items before being shuffled, thus deriving higher cost saving in shuffle-phase.

Fig.9(b) shows the performance improvement rates of HIP and IP for processing *Job2* in *Query2*. In HIP, summary information is always generated by children jobs (e.g., *Job1* and *Job2* in Fig.6(b)), and transported to parent jobs (e.g., *Job3* in Fig.6(b)) in a dataflow graph. Hence, in the case of HIP, *Job2* is accompanied with an overhead to generate summary information causing less than 1% slowdown. On the other hand, the IP approach can prune unnecessary data by using summary information transported from a sibling job (e.g., *Job1* in Fig.6(b)) in a dataflow graph. Hence, IP derives benefit in the output materialization step of *Job2* as shown in Fig.9(b). However, the benefit is relatively small (less than 1% speedups) for all block sizes.

Fig.9(c) compares the performance improvement rates of HIP and IP for *Job3* in *Query2*. First of all, the IP approach outperforms HIP for all block sizes. HIP showed more than 20% performance improvement by preventing unnecessary data before data shuffle-phase. On the other hand, IP achieved relatively better performance enhancements than HIP since already reduced intermediate data could reduce the cost in data loading phase in addition to the data shuffle cost. In the second place, with 512MB to 1024MB block sizes, both approaches showed almost constant performance improvements. Hive and HIP scheduled same numbers of mappers (86 mappers) and reducers (26 reducers) with those block sizes where the amounts of input data to each mapper and each reducer were not changed even if the block size increased. Hence, the effect of summary information in shuffle-phase was almost same with those block sizes. In case of IP, the same numbers of mappers (86 mappers) and reducers (14 reducers) were scheduled from 512MB to 1024MB block sizes. Therefore, the effect of reduced input data was almost same in data-loading and shuffling steps with different block sizes.

Performance measurements with TPC-H benchmark. Experiments with TPC-H benchmark were performed to measure the effectiveness of the IP approach. In these experiments, the bloom-filter size of IP and the

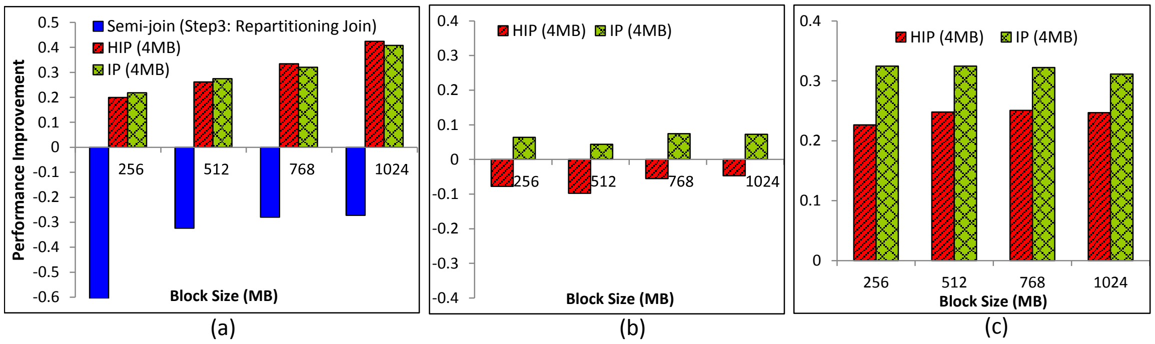


Figure 9: Performance improvement with varying block sizes (a) Query1:Job2 (b) Query2:Job2 (c) Query2:Job3

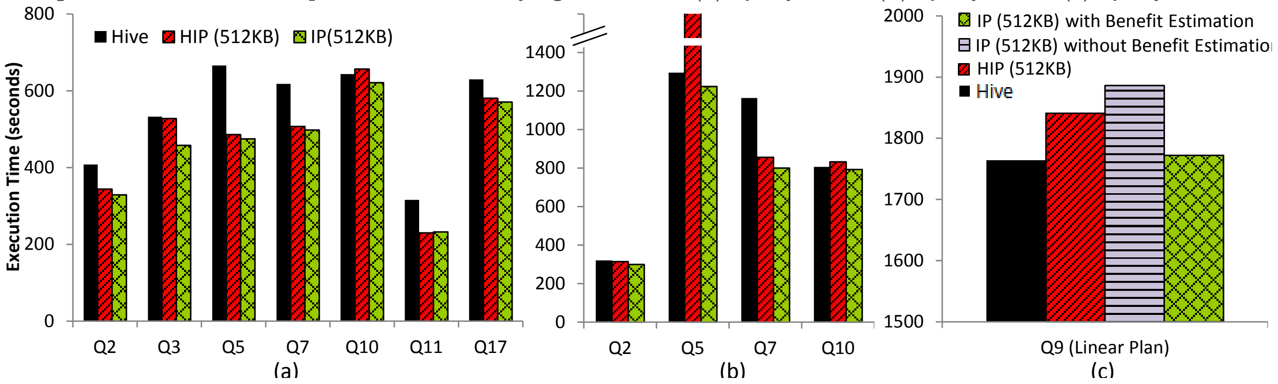


Figure 10: TPC-H queries with (a)linear and (b) non-linear plans, (c) Benefit estimation (TPC-H Q9 linear plan)

summary size limitation of HIP were configured to 512KB. Fig.10(a) shows the execution times of the different approaches for TPC-H queries which are translated into linear plans. HIP and IP approaches improve the query processing performance for most of the queries up to 27.0% and 28.6%, respectively. However, HIP does not improve the performance of *Q3*, and degraded the performance of *Q10* a little due to overhead of generating and transporting summaries. With non-linear plans as shown in Fig.10(b), the IP approach showed 5-6% performance improvements when processing Q2, Q5, and Q10, and improved the execution time of Q7 by about 31%. In HIP, the execution times of Q2 and Q7 were improved 2% and 26%, respectively while that of Q10 was degraded about 3% due to the overhead of summary generation. It is notable that HIP did not work for Q5 because one of the jobs in the plan did not have enough heap memory space for storing its output summary information. HIP stores a list of hash values on a join column in memory. Hence, if the size of the in-memory list exceeds available memory space, it drives reducers to fail their execution. On the other hand, IP stores such data in a more compact bloom filter whose size is configurable, thus avoiding such problems.

Benefit Estimation. HIP and IP approaches without benefit estimation, may worsen query processing performance for cases where the overhead of generating and transporting summary information exceeds the benefit achieved by using the summary information. For example, with TPC-H Q9, both approaches showed worse performance when they were enabled. Fig.10(c) shows the execution times of a Q9 linear plan in different approaches. We compared the original Hive, HIP, IP without benefit estimation, and IP with benefit estimation. In HIP, jobs always generate summary information as long as they have subsequent jobs to which such summary information can be transferred. In addition, it transports summary information to the next job if its size is less than a user-defined threshold, even if the summary may not be beneficial. Because of such summary generation and transmission overheads, HIP brought about a 80 second delay in the execution time of the Q9 linear plan. In IP without benefit estimation, all jobs in the query plan always generate summaries and transport them to next jobs. As a result, such imprudent use of summary information added a cost of about 125 seconds to the original execution time. On the other hand, the IP approach which leverages benefit estimation, could selectively allow summary generations and transmissions based on cost estimations. This allowed for maximizing the benefit of IP and preventing any large performance degradation in worst cases.

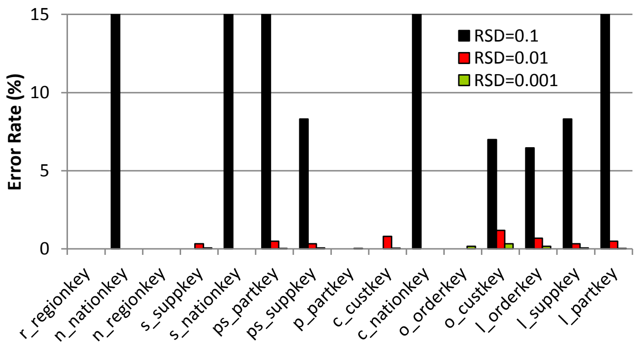


Figure 11: Accuracy for the number of distinct values with different RSDs

Piggyback Statistics Collection. Statistics collection which piggybacks query processing may impose penalties on its performance. This section evaluates the effect of piggy-

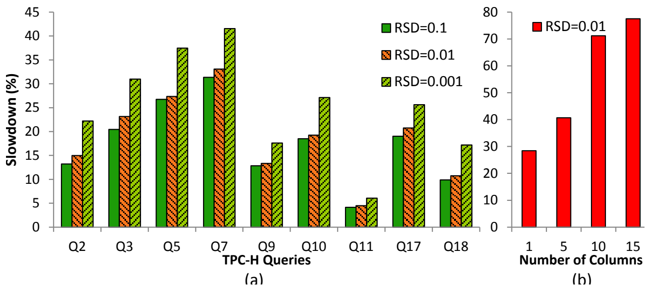


Figure 12: Piggyback statistics collection overhead (a) varying RSD values (b) varying number of columns

back statistics collection. While processing TPC-H queries, distinct value cardinalities were generated on columns that were either join keys or were involved in filter conditions. The slowdown on execution time was then measured. The slowdown for each query was calculated as following:

$$slowdown(\%) = \frac{T_P - T_O}{T_O} \times 100$$

where T_P is the execution time of a query with piggyback statistics collection, and T_O is the execution time of the original Hive approach. During the experiments, three relative standard deviations (RSDs) (0.1, 0.01, and 0.001) were used as a parameter to HyperLogLog. This was done to change the size of required memory space. As the RSD value increases, HyperLogLog requires less memory space while causing higher error rate. Fig.12(a) shows the slowdowns of the TPC-H queries. As RSD became smaller, the slowdown of query processing performance increased. When RSD = 0.001, slowdowns were between 6% and 41% while error rates in distinct value cardinalities were less than 0.4%. RSD = 0.01 caused 4-33% slowdowns with less than 1.6% error rates. With RSD = 0.1, slowdowns were between 4% and 31% with error rates less than 30.7%. Fig.11 shows error rates in distinct value cardinality estimation on a set of key columns. In the experiment, processing overhead to generate statistics was the main factor that affected the slowdowns of query execution times. For TPC-H Q7, the processing overhead for statistics generation affected about 97% of the slowdowns when RSD is 0.1 and 0.01, while registering partial statistics caused 3% of the slowdowns. However, when RSD = 0.001, the overhead to register partial statistics to the statistics repository caused about 19% of the slowdown. As a result, as the RSD value increases, partial statistics registration overhead can be one of the dominant factors that affect query performance.

Next, slowdowns were measured by manually changing the number of columns on which distinct value cardinalities were generated. This experiment used a query which loads *lineitem* table and performs a groupby operation on *l_linenumber*. RSD was fixed at 0.01. As shown in Fig.12, the increase in number of columns involving distinct value cardinality collection, resulted in more processing and statistics transmission overheads, thus increasing slowdown in query processing time. Hence, choosing minimal columns from which distinct value cardinalities are collected is necessary to decrease the performance degradation of piggyback statistics collection.

5. CONCLUSIONS

We present an adaptive information passing approach for early pruning of intermediate states in a MapReduce data processing workflow. The approach is based on a MapReduce-aware cost model for estimating potential benefits or loss of information passing in a particular workflow. We also present a light-weight approach for computing statistics by piggybacking on operator execution. Our approach for integrating the proposed information passing technique into a popular platform, Apache Hive, is presented. A comprehensive empirical evaluation using two datasets shows the benefits of our approach over existing techniques.

6. ACKNOWLEDGMENTS

This work was partially funded by NSF grant IIS-0915865 and IIS-1218277.

7. REFERENCES

[1] Apache hadoop. http://hadoop.apache.org.
[2] Apache hive. http://hive.apache.org.
[3] Hadoop 0.20 documentation. http://hadoop.apache.org/common/docs/r0.20.0/.
[4] Running the tpc-h benchmark on hive. https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf.
[5] Tpc-h. http://www.tpc.org/tpch/.
[6] Virtual computing lab. http://vcl.ncsu.edu.
[7] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.
[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.
[9] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[10] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD Conference*, pages 975–986, 2010.
[11] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.
[12] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD Conference*, pages 287–298, 2004.
[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
[14] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA*, pages 127–146, 2007.
[15] R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, pages 486–497, 2012.
[16] S. Hong and K. Anyanwu. Hip: Information passing for optimizing join-intensive data processing workloads on hadoop. In *DEXA (2)*, pages 384–391, 2012.
[17] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, pages 774–783, 2008.
[18] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.
[19] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD Conference*, pages 103–114, 1994.
[20] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, 2009.
[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[23] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 12:1–12:13, 2011.

To Auto Scale or not to Auto Scale

Nathan D. Mickulicz
YinzCam, Inc. / Carnegie Mellon University

Priya Narasimhan
YinzCam, Inc. / Carnegie Mellon University

Rajeev Gandhi
YinzCam, Inc. / Carnegie Mellon University

Abstract

YinzCam is a cloud-hosted service that provides sports fans with real-time scores, news, photos, statistics, live radio, streaming video, etc., on their mobile devices. YinzCam's infrastructure is currently hosted on Amazon Web Services (AWS) and supports over 7 million downloads of the official mobile apps of 40+ professional sports teams and venues. YinzCam's workload is necessarily multi-modal (e.g., pre-game, in-game, post-game, game-day, non-gameday, in-season, off-season) and exhibits large traffic spikes due to extensive usage by sports fans during the actual hours of a game, with normal game-time traffic being twenty-fold of that on non-game days.

We discuss the system's performance in the three phases of its evolution: (i) when we initially deployed the YinzCam infrastructure and our users experienced unpredictable latencies and a large number of errors, (ii) when we enabled AWS' Auto Scaling capability to reduce the latency and the number of errors, and (iii) when we analyzed the YinzCam architecture and discovered opportunities for architectural optimization that allowed us to provide predictable performance with lower latency, a lower number of errors, and at lower cost, when compared with enabling Auto Scaling.

1 Introduction

Sports fans often have a thirst for real-time information, particularly game-day statistics, in their hands. The associated content (e.g., the game clock, the time at which a goal occurs in a game along with the players involved) is often created by official sources such as sports teams, leagues, stadiums and broadcast networks.

From checking real-time scores to watching the game preview and post-game reports, sports fans are using their mobile devices extensively [11] and in growing numbers in order to access online content and to keep up to date on their favorite teams, according to a 2012 report from Burst Media [10]. Among the surveyed sports fans, 45.7% said that they used smartphones (with 31.6% using tablets) to access online sports-content at least occasionally, while 23.8% said that they used smartphones (with 17.1% using tablets) to watch sporting events live. This trend prevails despite the presence of television– in fact, fans continued to use their mobile devices to check online content as a second-screen or third-screen viewing-experience even while watching television.

YinzCam started as a Carnegie Mellon research project in 2008, with its initial focus being on providing in-venue replays and in-venue live streaming camera angles to hockey fans inside a professional ice-hockey team's arena [5]. The original concept consisted of a mobile app that fans could use on their smartphones, exclusively over the in-arena Wi-Fi network in order to receive the unique in-arena video content. While YinzCam started with an in-arena-only mobile experience, once the research project moved to commercialization, the resulting company, YinzCam, Inc. [15], decided to expand its focus beyond the in-venue (small) market to include the out-of-venue (large) market.

YinzCam is currently a cloud-hosted service that provides sports fans with real-time scores, news, photos, statistics, live radio, streaming video, etc., on their mobile devices anytime, anywhere, along with replays from different camera angles inside sporting venues. YinzCam's infrastructure is currently hosted on Amazon Web Services (AWS) and supports over 7 million downloads of the official mobile apps of 40+ professional sports teams and venues within the United States.

Given the real-time nature of events during a game and the range of possible alternate competing sources of online information that are available to fans, it is critical for YinzCam's mobile apps to remain attractive to fans by exhibiting low user-perceived latency, a minimal number of user errors (visible to user in the form of time-outs occuring during the process of loading a page), and

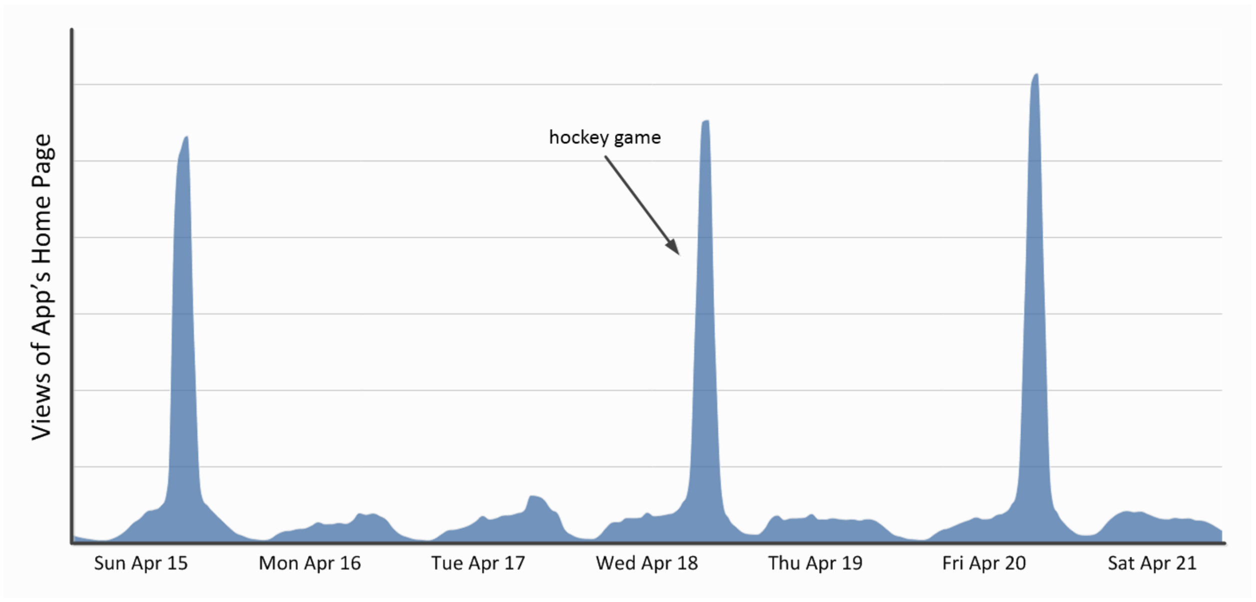


Figure 1: The trend of a week-long workload for a hockey-team's mobile app, illustrating modality and spikiness. The workload exhibits the spikes due to game-day traffic during the three games in the week of April 15, 2012.

real-time information updates, regardless of the load of the system. Ensuring *a responsive user experience* is the overarching goal for our system. Our infrastructure needs to be able to handle our *unique spiky workload*, with game-day traffic often exhibiting a twenty-fold increase over normal, non-game-day traffic, and with unpredictable game-time events (e.g., a hat-trick from a high-profile player) resulting in even larger traffic spikes.

Contributions. This paper describes the evolution of YinzCam's production architecture and distributed infrastructure, from its beginnings three years ago, when it was used to support thousands of concurrent users, to today's system that supports millions of concurrent users on any game day. We discuss candidly the weaknesses of our original system architecture, our rationale for enabling Amazon Web Services' Auto Scaling capability [2] to cope with our observed workloads, and finally, the application-specific optimizations that we ultimately used to provide the best possible scalability at the best possible price point. Concretely, our contributions in this paper are:

- An AWS Auto Scaling policy that can cope with such unpredictably spiky workloads, without compromising the goals of a responsive user experience;
- Leveraging application-specific opportunities for optimization that can cope with these workloads at lower cost (compared to the Auto Scaling-alone approach), while continuing to meet the user-experience goals;
- Lessons learned on how Auto Scaling can often mask architectural inefficiencies, and perform well (in fact, too well), but at higher operational costs.

The rest of this paper is organized as follows. Section 2 explores the unique properties of our workload, including its modality and spikiness. Sections 3, 4, and 5 describe our Baseline, Auto Scaling, and Optimized system configurations, respectively. Section 6 presents a performance comparison of our three system-configurations. Section 7 describes related work. Finally, we conclude in section 8.

2 Motivation

In this section, we explore the properties our workload in depth, describing the natural phenomena that cause its modality and spikiness. First, let's consider the modal nature of our workloads. Each workload begins in the non-game mode, where the usage is nearly constant and follows a predictable day-night cycle. In Figure 1, this can be seen on April 16, 17, 19, and 20. On game days, such as April 15, 18, and 20, our workload changes considerably. In the hours prior to the game event, there is a slow build-up in request throughput, which we refer to as pre-game mode. As the game-start time nears, we typically see a load spike where the request rate increases rapidly as shown in Figure 2. The system now enters in-game mode, where the request rate fluctuates rapidly throughout the game. In the case of hockey games, these fluctuations define sub-modes where usage spikes during

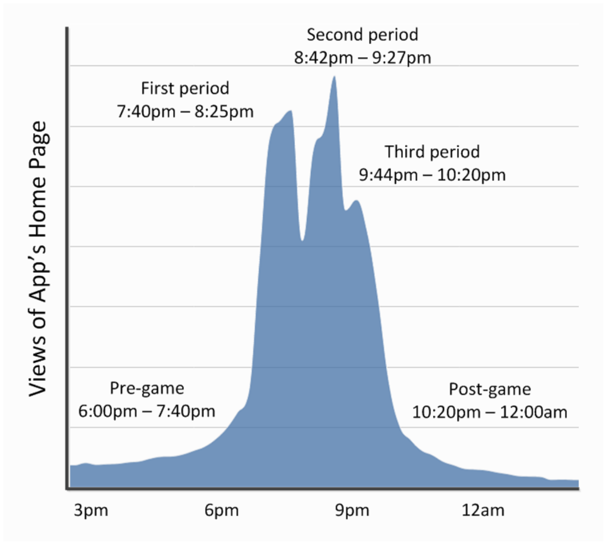


Figure 2: The trend of a game-time workload for a hockey-team's mobile app, illustrating modality and spikiness. The workload shown is for a hockey game in April 2012.

the first, second, and third hockey periods and drops during the 20-minute intermissions between periods. The load drops off quickly following the end of the game during what we call the post-game mode. Following the post-game decline, the system re-enters non-game mode and the process repeats.

In addition to modality, our workload exhibits what we call spikiness. We define a spiky workload to be one where the request rate more than doubles within a 15-minute period. The workload shown in Figure 2 exhibits spikiness at 7:30pm, towards the end of the pre-game phase. Between 7:30 and 7:45, the request rate of our workload more than doubled, and it nearly doubled again by 8:00pm. The request rate reached its maximum between 9:15 and 9:30 in the middle of the second period, when several significant scoring-events occurred. In addition to the rapid increases, our workload also shows equally-rapid decreases as the system enters the post-game mode, with the workload nearly halving in request rate between 10:15pm and 10:30pm.

To understand why our workload has these properties, we have to consider our users' demand for app content. Both figures 1 and 2 show the number of home-screen views in one of our hockey-team apps. Typically, this page shows news and media items as well as the box score of the previous game played. However, during games, the home screen shows real-time data such as live team-statistics and the game clock as well as a portion of team's Twitter feed. This effectively provides fans with a way to follow the game without being tuned in via radio or television. It also provides a second screen that can augment an existing radio or television broadcast with live statistics and Twitter updates. Our fans have found these features to be tremendously useful and compelling, and our workload shows that these features are used heavily during periods when the game is in play. At other times, the app provides a wealth of information about the team, players, statistics, and the latest news. These functions, which lack a real-time aspect, have their usage spread evenly throughout the day.

3 Configuration 1: Baseline

Figure 3 shows the architecture of our system when we first began publishing our mobile apps in 2010. The system is composed of two subsystems using a three-tier architecture, one consuming new content and the other pushing views of this content to our mobile apps. The two subsystems share a common relational database, shown in the middle of the diagram.

The content-aggregation tier is responsible for keeping the relational database up-to-date with the latest app-content, such as sports statistics, game events, news and media items, and so on. It runs across multiple EC2 instances, periodically polling information sources and identifying changes to the data. The content-aggregation tier then transforms these changes into database update queries, which it executes to bring the relational database up-to-date. Since these updates are small and infrequent, the load on the EC2 instances and database is negligible. The infrequency of updates also allows us to use aggressive query-caching on our app servers, preventing the database from becoming a bottleneck.

Our apps periodically retrieve new information from the content-storage tier in the form of XML documents. The app then displays information from the XML document to the user in various ways. The content-delivery tier is responsible for composing views of the relational data as XML documents that are ready for consumption by our apps. In response to a request for XML data, the content-delivery tier executes one or more database queries and synthesizes the results into an XML document on the fly. This task is both I/O-intensive and CPU-intensive. Fortunately, scaling up this component is simply a matter of adding additional EC2 instances behind a load balancer.

However, despite being conceptually simple, we encountered multiple problems scaling up our system as the size of our user-base increased. Initially, scaling up the content-delivery tier was an entirely-manual task. We did this before every sporting event, and we could only guess how many additional servers we would need to handle the CPU load for the game. Since the spikes we see are of varying magnitude, we would often provision too

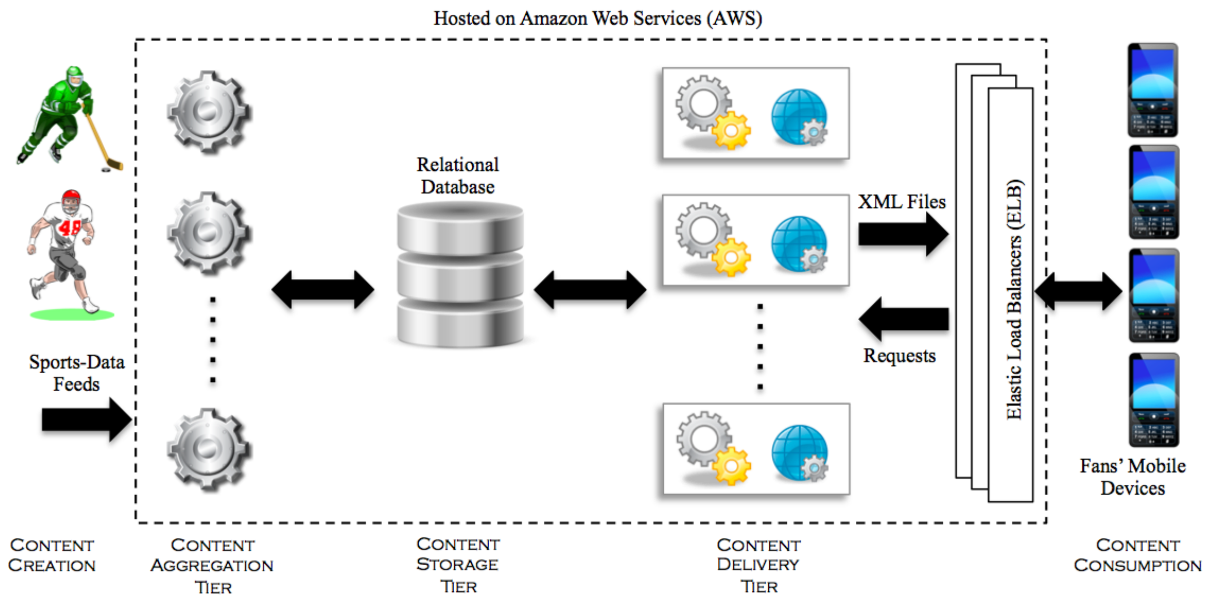


Figure 3: Baseline Configuration

much or too little, either wasting money on resources we didn't need or frustrating our users with high latency and errors. In the results section, we describe the high response latency of our baseline architecture when under-provisioned for a game workload.

4 Configuration 2: With Auto Scaling

We first adopted EC2 Auto Scaling [2] to deal with the CPU-load-management problem. Auto Scaling is a technique that allows system administrators to automatically scale a fleet of EC2 resources up or down to meet current workload-demands. We defined an Auto Scaling policy that allowed our system to adapt to the spikes in our workload.

We follow an aggressive scale-up policy in our system to cope with our spiky workloads. Spikes in our workload happen very quickly, and if we do not scale our resources quickly, many of our users experience high latency and errors while additional resources are being added. We use current CPU-usage in our content-delivery tier to determine when to add additional servers. We have set the CPU-usage threshold for scale up to a low value of 30% average over 1 minute, in an attempt to catch spikes early. We also scale up by doubling the size of the fleet to make sure that we have enough instances available to handle double the workload volume.

On the other hand, we defined a scale-down policy that was slow and cautious. There are periods of lower-volume usage (such as between periods in a hockey game) where we did not want to scale down prematurely. Furthermore, scaling down too rapidly could remove too many resources from the pool, forcing the system to immediately scale up after the next CPU-utilization check. This could cause the system to flap back and forth between two fleet sizes, wasting EC2 resources.

We downscale our fleet of servers by removing one server at a time and making sure that the CPU usage in our content-delivery tier is stable for a longer period of time before we do another round of downscaling. Thus, our Auto Scaling policy can be described as Multiplicative-Increase-Linear-Decrease (MILD). This policy was inspired by the Additive-Increase-Multiplicative-Decrease (AIMD) policy for congestion control used in TCP. As with congestion control, our scaling policy attempts to prevent load-collapse by cautiously modifying parameters of the system until the workload matches the system's capacity. The major difference between MILD and TCP's AIMD is that TCP gradually increases the workload until the network is at capacity. Since we do not have the ability to rate-limit our workload, we take the opposite approach of gradually reducing our system's maximum capacity until this capacity matches the workload.

As described in section 6, Auto Scaling does solve the high-latency problem caused by high CPU load for the baseline configuration. Adding additional instances effectively adds more CPU resources, and when placed behind a round-robin load-balancer such as Amazon's ELB [3], each instance gets an equal share of the workload. Furthermore, Auto Scaling only increases the size of the fleet when the workload demands it, so we don't have to over-provision for each game. With Auto Scaling, we were able to scale up our system to get acceptable latencies in the face of our spiky workload even though our system had a sub-optimal design with glaring inefficiencies.

Unfortunately, masking inefficiencies with Auto Scaling does not come without a cost. We had to pay for up to 15 additional instances per team during each game, which adds up to a considerable increase in operation costs over an entire season of games (for our hockey apps, 82 regular-season games per team). At this point, we wondered if we could lower our operations costs by removing the inefficiencies in our architecture, thus lowering our CPU requirements and the number of instances required to handle our workload during games. The subsequent analysis of the inefficiencies in our system led us to the optimized architecture we describe next.

5 Configuration 3: Optimized

Our optimized architecture is the result of studying our baseline architecture, identifying inefficiencies, and modifying the architecture to correct them. After studying our system, we identified two major problems with our architecture. The first problem was in our request handling, where we realized that every request required the server to generate a new XML document from data stored in the database. Often, the system generated multiple identical XML documents within a short period of time. The second was in our database layer, where we noticed that certain queries were being executed multiple times within a few seconds, each returning the same result. These observations led us to add two caching-layers to our architecture, which we describe below.

In response to the observation that every request required XML generation, we added a caching layer in front of the content-delivery tier. This layer receives requests from clients and serves pre-generated XML content if the cache time on the content has not expired; otherwise, it regenerates the XML content using the content-delivery tier and stores the content to serve subsequent requests. This dramatically reduces instance CPU utilization, since new pages are only generated when cached content expires (instead of on every request).

We implemented our caching layer using the output-caching feature of the Microsoft IIS web server, which required very little additional code or configuration on top of our existing IIS-based content-delivery tier. We assigned groups of XML documents a cache-expiration time based on how much staleness the content could tolerate. For example, XML documents describing a news article are unlikely to change after publication and so have a cache-expiration time of a day or more, while documents describing the latest game events change frequently and have cache-expiration times on the order of seconds.

Our second optimization was another caching-layer between the content-delivery tier and the content-storage tier, in response to the observation that the content-delivery tier was executing multiple queries for the same data in rapid succession. While our relational database has an internal query cache, some of our queries generate tens of megabytes of data. When several of these queries are run in parallel, this volume of data can quickly saturate the network link between our EC2 instances and the database. To remedy this, we added a cache of query results in the content-delivery tier.

This cache is implemented in a similar fashion to the output cache. When a server in the content-delivery tier needs to execute a database query, it first looks up the query in a table of query results. If the query is found in the table and the result has not expired, the result from the table is used to generate the XML. Otherwise, the generator runs the database query directly and updates the table with the results. Like individual XML-documents, each query is assigned a cache time based on the staleness-tolerance of the data.

6 Evaluation

We evaluated our three system architectures using a HTTP load generator and a trace of a production workload. We ran the three-hour trace and recorded the average latency seen over consecutive 60-second intervals. We then compared the results of these tests to determine the effectiveness of each approach in reducing the response latency seen by our users.

We built our testbed entirely on EC2, utilizing EC2's internal network for communication between simulated clients and the view generation service. Using the logs collected during the April 18, we simulated clients by using a program that replays the log file exactly as it was recorded using the timestamps associated with each request. The EC2 instance used to simulate clients was of type m1.xlarge.

We ran our tests against a copy of our production system. This copy was set up in an identical configuration as our production system, except it was not serving production traffic. To make this copy, we duplicated both the load balancer configuration and the Auto Scaling configuration. We used the same operating system image as the production service with the same EC2 instance type (c1.medium). We also created an isolated read replica of our database used only by the EC2 instances involved in the test.

Our test trace was a three-hour log of traffic served by our production system during a hockey game in April

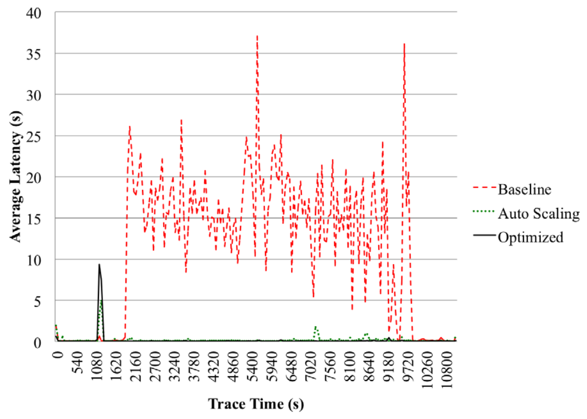


Figure 4: The average latency of our three system configurations throughout a 3-hour production workload trace. The workload was recorded during a hockey-game in April 2012.

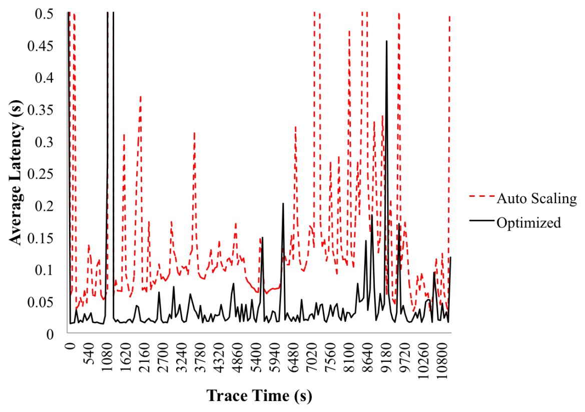


Figure 5: The same results shown figure 4, zoomed in to show the differences between the Auto Scaling and Optimized configurations.

2012. This trace captures all of the spikes in our workload. For each configuration of our system, we replayed this trace in real time using our client simulator. Each test was run in isolation on our test infrastructure, and all resources were rebooted between runs.

We tested three configurations of our system: Baseline, Auto Scaling, and with Optimizations. The Baseline configuration is the system described in section 3, running on a single instance through the entire trace. The Auto Scaling configuration is the Baseline configuration running with the Auto Scaling policies described in section 4, configured with one initial instance but a maximum of sixteen. Finally, the Optimized configuration is the system described in section 5, which extends the Baseline configuration with additional caching to improve performance. Like the Baseline configuration, the Optimized configuration is restricted to a single instance.

Figure 4 shows a comparison of average latency over 60-second windows throughout the entire 3-hour trace across all three configurations. The baseline configuration performs much worse than either the Auto Scaling or Optimized, due to the restricted amount of CPU time available on a single instance. Both Auto Scaling and Optimized perform far better through their respective mechanisms for coping with heavy load - Auto Scaling simply adds more resources to the pool, while Optimized makes more efficient use of CPU resources.

Figure 5 shows the same trace comparison as in figure 4, except zoomed-in to show the fine differences between the Auto Scaling and Optimized cases. This comparison shows that Optimized has both lower average-latency and lower jitter than the Auto Scaling case throughout the entire trace. Through careful optimization, we were able to outperform Auto Scaling using just a fraction of the eight instances that Auto Scaling required to achieve comparable performance.

7 Related Work

A substantial amount of research has been done in the area of using Auto Scaling to cope with workloads that have a high peak-to-average ratio. A number of related works use predictive models to forecast the workload and then use Auto Scaling to dynamically adjust the resource usage to match the predicted workload. The main difference between some of these proposals is the way predictive models are built and used. For instance, [12], [1], [8] use control-theoretic models to predict workloads while [7], [13] use autoregressive moving average (ARMA) filters for prediction. Mao et. al. [9] use Auto Scaling to scale up or down cloud infrastructure to ensure that all job deadlines are met under a limited budget. There is also a significant amount of research done in the area of workload modeling and prediction, even though the proposed systems do not explicitly mention using Auto Scaling to cope with varying work load. Gmach et. al. [6] use resource pools that are shared by multiple applications and propose a process for automating the efficient use of such resource pools. Shivam et. al.[14] use an active-learning approach to analyze the performance histories of hosted applications to build predictive models for future use of the applications and use the predicted values for future resource assignment. Chandra et. al. [4] propose to capture the transient behavior of web applications workload by modeling the server resource.

While we can certainly benefit from some of the workload-prediction-related work, as we demonstrate in Section 2, our workloads can be spiky and are often hard to predict in advance (e.g. predicting whether a player will score a hat-trick). We believe that contextual prediction e.g., prediction based on the analysis of statistics feeds to determine what might be going on in the game (a player close to scoring a hat-trick), or prediction based on the analysis of news feeds to learn about important events (like return of a favorite player), might be more appropriate and useful in our context.

8 Conclusions and Reflections

Auto Scaling is often provided by IaaS providers as a technique by which applications can scale up/down resources to meet the current demand. Our results show that Auto Scaling works well, in-fact so well that we were able to take a system with obvious architectural flaws and make it perform nearly as well as a fully-optimized version. However, hiding these inefficiencies comes with the price of additional infrastructure. In our case, our inefficient Baseline-configuration required up to eight times the resources of our efficient Optimized-configuration to achieve comparable performance.

On the other hand, the optimizations that we made were fairly simple and straightforward to identify and implement. This may not be the case with all systems, and optimizing effectively often requires considerable skill and effort. However, if it can be done, the payoff can be much greater than simply using Auto Scaling with an inefficient system. In our case, our thoughtful optimizations required greater insight and more development time, but paid off through lower costs, lower latency, and lower jitter than either of the other configurations.

Acknowledgments

The authors would also like to thank Justin N. Beaver and Michael P. Kasick for their contributions to YinzCam's cloud-hosted infrastructure and services.

References

[1] Abdelwahed, S., Bai, J., Su, R., and Kandasamy, N. On the application of predictive control techniques for adaptive performance management of computing systems. *IEEE Transactions on Network and Service Management 6* (Dec. 2009), 212–225.

[2] Amazon Web Services. Auto Scaling. "http://aws.amazon.com/autoscaling/".

[3] Amazon Web Services. Elastic Load Balancing.

[4] Chandra, A., Gong, W., and Shenoy, P. Dynamic resource allocation for shared data centers using online measurements. In *ACM/IEEE International Workshop on Quality of Service (IWQoS)* (2003).

[5] ComputerWorld. Context on ice: Penguins fans get mobile extras. "http://www.computerworld.com/s/article/9134588/Context_on_ice_Penguins_fans_get_mobile_extras".

[6] Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A. Capacity management and demand prediction for next generation data centers. In *IEEE International Conference on Web Services* (2007), pp. 43–50.

[7] Gong, Z., Gu, X., and Wilkes, J. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management* (2010), pp. 9–16.

[8] Li, J., J. Chinneck, M. Woodside, M. L., and Iszlai, G. Perfomance model driven QoS guarantees and optimization in clouds. In *ICSE Workshop on Software Engineering Challenges of Cloud Computing* (2009), pp. 15–22.

[9] Mao, M., Li, J., and Humphrey, M. Cloud auto-scaling with deadline and budget constraints. In *IEEE/ACM International Conference on Grid Computing* (2010), pp. 41–48.

[10] Mobile Marketer. 45.7% of sports fans use smartphones to access content online. "http://www.mobilemarketer.com/cms/news/research/14020.html".

[11] Online Media Daily. Super sports fans engage on mobile. ="http://www.mobilemarketer.com/cms/news/research/14020.html".

[12] Padala, P., Shin, K., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. Adaptive control of virtualized resources in utility computing environments. *ACM Operating Systems Review 41* (June 2007), 289–302.

[13] Roy, N., Dubey, A., and Gokhale, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE International Conference on Cloud Computing* (2011), pp. 500–507.

[14] Shivam, P., Babu, S., and Chase, J. S. Learning application models for utility resource planning. In *International Conference on Autonomic Computing* (2007), pp. 255–264.

[15] YinzCam, Inc. "http://www.yinzcam.com".

Big Data Exploration via Automated Orchestration of Analytic Workflows

Alina Beygelzimer
IBM T. J. Watson Research Center

Anton Riabov
IBM T. J. Watson Research Center

Daby Sow
IBM T. J. Watson Research Center

Deepak S. Turaga
IBM T. J. Watson Research Center

Octavian Udrea
IBM T. J. Watson Research Center

Abstract

Large-scale data exploration using Big Data platforms requires the orchestration of complex analytic workflows composed of atomic analytic components for data selection, feature extraction, modeling and scoring. In this paper, we propose an approach that uses a combination of planning and machine learning to automatically determine the most appropriate data-driven workflows to execute in response to a user-specified objective. We combine this with orchestration mechanisms and automatically deploy, adapt and manage such workflows across Big Data platforms. We present results of this automated exploration in real settings in healthcare.

1 Introduction

With the emergence of multiple Big Data platforms that handle large volumes of streaming and stored data, it is becoming possible to support massive data exploration tasks in many different domains. These domains include cybersecurity, healthcare, financial services, manufacturing process control, as well as several environmental monitoring applications. More specifically, in intensive care, healthcare providers need large-scale and real-time exploration of medical records, test results, and physiological data streams from monitored patients to detect complications as early as possible. There are several ways to explore this data for these detection problems. As a result, such an exploration requires constructing and orchestrating a large number of analytic flows, i.e. workflows composed of atomic analytic components for data selection, feature extraction, modeling, and scoring. The scale of data requires the use of a distributed setting, potentially across multiple compute platforms (e.g. offline learning on Hadoop and online scoring on a real-time stream computing platform). This places a near insurmountable burden on end users and analysts who want to utilize these platforms and analytics in their domain, and motivates the need for autonomic management of the analytic workflows.

In this paper we propose a solution based on autonomic computing principles for creating, deploying, self-managing and adapting analytic workflows in response to an end-user's high level specification of their objectives. Specifically, we propose an approach that combines planning and machine learning to automate the composition and choreography of these workflows in large-scale distributed data exploration tasks. The use of machine learning in autonomic compute systems has been explored previously in limited settings related to scheduling and resource management. In [14] [20] the authors use reinforcement learning for fairly scheduling resources in a large-scale production grid, and resource allocation in distributed settings, respectively. The use of planning in software composition with semantic constraints has been studied in web services [18]. We build on an existing planning-based composition tool, MARIO [15], which was originally created to allow end-users to compose and deploy analytics on multiple platforms.

Combinations of planning and learning have been used in robotics [3] for exploring and partitioning complex state spaces in noisy and stochastic settings, and for imitation learning [16]. However this work is primarily focused on exploring uncertain environments as opposed to analytic workflow composition, selection, and orchestration, as discussed in this paper. Planning and learning for analytic workflow selection has been recently studied in [10]. The authors use a data mining ontology to capture an algorithm's inductive bias, and use learning to select the algorithm to use in different settings. However, this does not consider the problem of constructing analytic workflows by dynamically composing such individual algorithms.

In contrast, in this paper, we focus on both the composition of analytic flows as well as the data-driven dynamic selection of appropriate flows to meet an end-user objective. By describing atomic analytic components with semantic annotations, we provide end users with the ability to specify their objectives using a business relevant semantic vocabulary. This objective is associated with an appropriate objective or loss function that may be used to evaluate performance. We use planning techniques to identify feasible analytic compositions in response to the user-specified goal. We then use online learning to iteratively explore the space of feasible compositions to determine the most appropriate workflows to deploy in a data-driven manner. We combine planning and learning with orchestration mechanisms that allow us to deploy and manage analytic workflows on a large-scale, real-time, distributed stream processing platform (IBM InfoSphere Streams) [19]. This automation allows us to adapt to dynamics in the data, availability of new analytic components, as well as system resource constraints, while providing predictive performance.

This paper is organized as follows. We describe the technical details of our approach and the underlying system architecture in Section 2, including the planning component 2.1 and the learning component 2.2. We then describe results of using this system on real-world exploration problems in a healthcare setting in Section 3 and highlight both predictive performance as well as system dynamics. We finally conclude with a discussion and directions for future work in Section 4.

2 System Description

Our design for an autonomic system for data exploration is based on three observations of the typical application scenario: (i) there are multiple analysis workflows that have different degrees of usefulness for a data analysis objective (and that degree may change over time); (ii) each such workflow is a combination of data sources and analytics, including feature extraction, model building and scoring components; (iii) available computing resources may limit the number of workflows (combinations) that can be in execution at any time.

Consider a healthcare application scenario [1] in which the objective is to predict complications in an ICU setting ahead of time. The available data includes offline data such as histories and outcomes of previous patients, the history of the current patient; slow changing data such as results of physician ordered tests; and live streaming data such as sensor measurements from the patient's monitor units (e.g., ECG, blood oxygen levels, respiration rate, etc.). The system also has available analytics that can extract both simple and complex features from this data, build a variety of machine learning models from this data (including but not limited to decision trees, SVMs, etc.). Compositions of these algorithms into workflows are required to solve the detection problem, however only some workflows are meaningful,

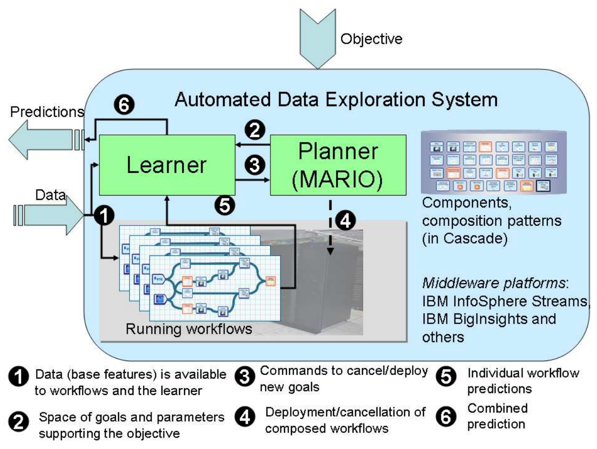

Objective
Automated Data Exploration System
Predictions
6
Learner
2
3
Planner (MARIO)
Data
1
5
4
Components, composition patterns (in Cascade)
Middleware platforms: IBM InfoSphere Streams, IBM BigInsights and others
Running workflows
1 Data (base features) is available to workflows and the learner
2 Space of goals and parameters supporting the objective
3 Commands to cancel/deploy new goals
4 Deployment/cancellation of composed workflows
5 Individual workflow predictions
6 Combined prediction

Figure 1: System architecture

and moreover the choice of workflow depends on context and varies over time. We use planning and learning to dynamically determine the most effective analytic workflow to construct and deploy given our computational resources, and the user specified task.

With these objectives in mind, we propose a design described in Figure 1. There are two main components of the system described in the remainder of this section. The *planner* has access to a repository of descriptions of analytic components and patterns available to the system. The analytic components are semantically annotated, and together with the constraints expressed in the patterns describe the space of analytic workflows that can be automatically composed. The purpose of the planner is to discover the set of goals and parameters that match a specific objective such as predicting patient complications, and for each such goal to automatically compose, generate code and deploy an analytic workflow that realizes the goal. A specific goal relevant in intensive care may be to detect ectopic or abnormal heart beats by observing electrocardiograms (ECG), identifying individual heart beats, extracting spectral features, and classifying them using a decision tree algorithm.

The *learner*'s mission is threefold: (i) over time, it learns the effectiveness of the various analytic workflows deployed for the objective; (ii) it makes a single prediction for specific complications as a function of the effectiveness of the analytic workflows and their individual predictions; (iii) it samples from the space of available analytic workflows that match our computational resources – the sampling is performed as a function of the learned effectiveness of workflows. The learner will continuously re-evaluate the current mix of analytic workflows deployed and, when deciding to change this set, communicate with the planner which will compose and deploy the analytic workflows, potentially on multiple platforms. The learner limits the total load on the system by limiting the number of flows that are active at any given time, and the middleware platforms further ensure efficient allocation of distributed computational resources to the flows selected by the learner.

In our implementation, the planner (including orchestration components) is running as a set of plugins in the Equinox OSGI container, while the learner component is running as a streams processing application on the IBM InfoSphere Streams platform. The two components communicate via a REST API over HTTP. We now describe these components in more detail.

2.1 Planning Component: MARIO

We rely on automated planning to adaptively determine the set of available analytic workflows given changing conditions, resources, data sources, data transforms, and analytics. Any new analytic added to our system is automatically combined with other compatible analytics to generate multiple new workflows. The planner also automatically eliminates inefficient workflows based on estimates of computational cost and reasoning about semantic equivalence of results. The remaining workflows are then made available for the learner to instantiate.

The planning, deployment orchestration, and development environment (for describing workflow composition constraints and semantics of analytics) in our implementation are provided by MARIO (Mashup Automation with Runtime Invocation and Orchestration) [15]. MARIO is responsible for:

1. Generating the complete set of distinct, efficient and valid analytic flows, given the set of analytics, data sources and composition patterns.
2. Generating platform-specific code and deploying individual selected flows with specified parameter values when instructed by the learner.

Our implementation is capable of generating code for IBM InfoSphere Streams and can be extended with plug-in code generators for other Big Data platforms, such as Apache Hadoop. In addition, MARIO provides a web application for end users, allowing them to inspect the results of running flows, predictions made by the learner, and request additional processing to be deployed.

SPPL planner MARIO uses a specialized planner to solve compositions as planning problems described in SPPL [17]. SPPL is derived from the general-purpose domain description language PDDL [8] and includes modifications to improve planning performance in workflow composition applications. In SPPL, description of the planning task includes description of planning domain, consisting of a set of actions with preconditions and effects defined as lists of user-defined predicates, and description of the planning problem containing the predicates of the initial state and the goal state. Given the planning task, the SPPL planner finds an optimal plan, i.e., a partially ordered set of actions that achieve the goal state when applied to the initial state, and optimize a linear objective subject to linear budget constraints.

Cascade composition patterns MARIO generates SPPL descriptions automatically based on composition patterns specified in Cascade [15] that describe composition constraints and software component semantics. Composition constraints are defined by defining a flow graph with points of variability and parameter ranges. Figure 2 includes an example pattern described in Cascade for a simple classification problem we used in experiments. The graph consists of two nodes, transform and classification, with two possible implementations of the transform. Ranges of parameters are specified as enumerations separated by a vertical bar "|". Implementations of *Classification*, *Features_DCT* and *Features_FFT* are defined separately in Cascade by providing platform-specific code fragments. Different choices of parameter values or implementations of the transform can be selected independently, thus generating many possible individual workflows based on the pattern.

In general, Cascade patterns can describe any directed acyclic graphs and can be recursive. For example, *Classification* can be defined as another pattern consisting of lower-level components. Individual components, i.e. analytics or data sources, can be implemented in any programming language supported by MARIO. Since MARIO only generates code for execution on target Big Data platforms and does not process any data itself, it places no restrictions on supported data types or the complexity of analytics. MARIO does not verify schema compatibility between connected components, and Cascade Developers have to ensure that the pattern enforces input/output compatibility of components.

2.2 Learning Component: Learner

At each step, the planner identifies the current set of feasible analytic flows, but it cannot tell which of these flows are useful for the current prediction problem. The goal of the learner is to automatically explore the space of feasible flows, learning the best current combination to deploy, subject to resource constraints. This is achieved in a data-driven way using feedback from the environment.

Our core learning algorithm is online gradient descent with several improvements, including adaptive feature-dependent gradient updates [6, 12] modified for loss nonlinearity as described in [11]. This approach works very

```
composite EctopicBeatDetectionPattern(output o) {
param
 string $NUM_PTS:
  UserParam("Number_Of_DCT_Features","|16|32|48|");
 string $MODELTYPE:
  UserParam("MODELLING_TYPE","|J48|NB|");
 string $MODELTRAINING:
  UserParam("NUMBER_OF_TRAINING_SAMPLES",
            "|500|1000|2000|");
graph
 stream transform =
     Features_DCT(){ param DCT_NUM_PTS: $NUM_PTS; }
   | Features_FFT(){ param FFT_NUM_PTS: $NUM_PTS; }
 stream o = Classification(transform){
   param CLASS_NUM_PTS: $NUM_PTS;
      MODELTYPE: $MODELTYPE;
      MODELTRAINING: $MODELTRAINING;  }
}
```

Figure 2: Example of a Cascade pattern.

well in high dimensional sparse feature spaces, typical in the applications we target.

We now describe the meta-learning problem (learning across other analytics which may themselves use learning), starting with the full information setting, when the learner can run and observe outputs of all available analytic flows at each step. Then we discuss the exploration problem arising from not being able to run most flows at every step due to resource constraints.

The meta-learner operates in an online setting where it repeatedly

1. receives input vector $\mathbf{x}_t \in \mathbb{R}^d$, which includes the outputs of all available analytic flows at time t,
2. makes its prediction $\hat{y}_t = \mathbf{w}_t \cdot \mathbf{x}_t$ of the target variable $y_t \in \mathbb{R}$, where $\mathbf{w}_t \in \mathbb{R}^d$ is the current linear model.
3. upon receiving feedback y_t, updates

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - 2\eta_t(\hat{y}_t - y_t)\mathbf{x}_t,$$

   where η_t is the learning rate at time t.

While the update rule above assumes the squared loss $\ell(\hat{y},y) = (\hat{y}-y)^2$ commonly used in regression, other loss functions are supported as well. The choice of the loss function is driven by the prediction problem. The gradient update actually used in the system is more complex, based on a combination of improvements described in detail in [6, 12, 11]. Other learning parameters of the learner that govern the form of the gradient, e.g. the learning rate can be optimized on a given problem using progressive validation loss [2, 4].

The analytic flow exploration problem requires extentions to this basic setting. Under resource constraints, when the number of feasible flows is large, they cannot be all instantiated together, hence the learner needs to carefully select which flows to run. Thus instead of observing the entire vector $\mathbf{x}$, the learner can only probe into it sparingly, observing only a small subset of values each time. Such attribute efficient learning for linear regression has recently been explored [5, 9], and we build on these for our exploration. While we omit details here, the intuition derives from the model in the basic setting. When properly normalized to account for the scale of flow outputs, the learned weight vector $\mathbf{w}_t$ indicates the relative importance of each flow. Flows whose weights are close to zero do not have much predictive edge for the current prediction problem. The learner's model $\mathbf{w}_t$ is continuously adapted to changes in the underlying data distribution, as illustrated in Section 3, and this model may be used (e.g. as a probability distribution) to control how the flows are sampled.

There are several other open problems for this exploration. First, we need to account for switching cost considerations associated with starting and stopping workflows. Second, we need a mechanism for learning new useful nonlinearities automatically from data. Finally, while in many applications, the loss function is known (as in classification or regression), there are scenarios, e.g. contextual bandit learning [7] where the loss function has to be learned as well. We are extending our learner component to tackle these open problems, and our results on these extensions will be described in detail in a separate publication.

3 Experimental Results

In this section, we describe two different types of illustrative results – results on a simulated example to highlight the adaptation and convergence of the learning based exploration, and some preliminary results on real-world datasets in healthcare.

3.1 Convergence and Adaptation Results

To illustrate the system's convergence and adaptation, we generated a regression dataset consisting of five features, $x_1,\ldots,x_5$, with values drawn independently at random from $[0,1]$ at every step. We then set the label at different time periods as shown in Table 1. We set $\alpha = \frac{t-t_s}{t_e-t_s}$. The experiment includes both sudden as well as gradual variations in the label characteristics. We create 15 analytic flows, that correspond to five self-products, and the ten pairwise products of features $x_1,\ldots,x_5$, and use a squared loss function $\ell(\hat{y},y) = (\hat{y}-y)^2$.

Figure 3 shows the corresponding instantaneous squared loss. The best constant's loss in this generated example was 0.1395, with the best constant predictor being 0.1675. The adaptive learner gives a relative improvement of 96.4% in squared loss in this case. Observe

Start(t_s)	**End**(t_e)	y
1	4000	$(x_1 - x_2)^2$
4001	6000	$(x_2 - x_3)^2$
6001	8000	$(1-\alpha)(x_2 - x_3)^2 + \alpha(x_4 - x_5)^2$
8001	10000	$(x_4 - x_5)^2$

Table 1: Generated Data.

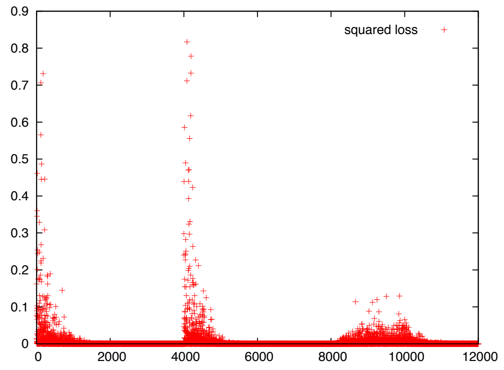


Figure 3: Instantaneous squared loss

the sharp increase in loss in step 4,000 when the target sharply changes to another function. The loss increases smoothly when the target starts to drift away from the learned function in step 8,000.

Figure 4 shows how the weights of different flows evolve in this experiment. The red curve corresponds to flow x_1^2, which is predictive only in the first 4,000 steps, while the target is $x_1^2 - 2x_1x_2 + x_2^2$. Its weight goes down to 0 when it no longer carries any predictive signal. The green curve corresponds to flow x_1x_2; the system learned to use it with coefficient -2 for the first target, and then the coefficient went down to 0 when the target changed to $x_2^2 - 2x_2x_3 + x_3^2$. The coefficients of features $x_1, \ldots, x_5$ are close to 0 for the entire experiment.

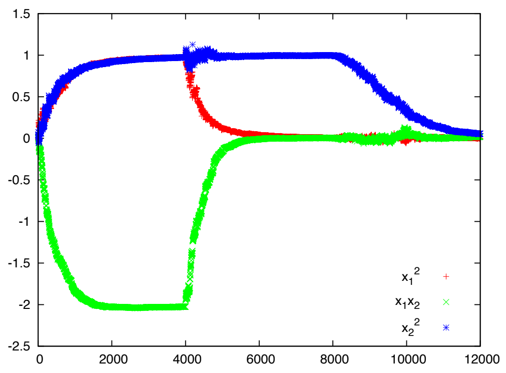


Figure 4: Weight evolution of different analytics

3.2 Exploring Healthcare Data

We consider an exploration problem in healthcare, focused on streaming analysis of Electro Cardiogram (ECG) signals from hospitalized patients. The application is focused on identifying *ectopic* or irregular heartbeats, that are indicative of potential problems to the patient. Detecting such beats from ECG signals may be viewed as a binary classification problem, and requires online learning to adapt to the time-varying nature of the input signals that vary with patient state, medications etc.

We make use of annotated data from the MIT Physiobank [13], a database with around 135000 annotated heartbeats (around 16000 ectopic)- corresponding to raw ECG data from 47 patients. The analytic workflow required for ectopic beat detection includes heartbeat and feature extraction, followed by binary classification. The Cascade pattern is shown in Figure 2. There is a choice between different transformations (DCT or FFT) with retention of only `$NUM_PTS` (16, 32, 64) coefficients, for feature extraction.

As classifiers, we use Weka [21] implementations of Decision Trees (J48) and Naive Bayes (NB) with periodic retraining - controlled by parameter `$MODELTRAINING` (500, 1000, 2000), i.e. retraining of the model is performed every `$MODELTRAINING` heartbeats. Hence, the resulting analytic flow space includes 36 combinations (2 transforms, 2 classifiers, and 3 values each for `$NUM_PTS`, and `$MODELTRAINING`).

These flows are deployed on the IBM InfoSphere Streams processing platform. We deploy a separate data source and feature extraction job for each patient, with a common set of classifier flows for all patients that are selected dynamically by the interaction between the learner and the planner. We deploy these jobs across a cluster with 8 compute nodes with 8 cores each and a shared filesystem.

We compare the prediction performance (in terms of probability of detection p_D and probability of false alarm p_F) of this automated deployment with the best hand-tuned deployment, achieved after multiple man-hours of experimenting and tuning in batch mode. We also include results to indicate the impact of different resource constraints on the deployment. These are shown in Table 2. For each automated experiment, we were allowed to deploy a maximum of 5 analytic flows at one time. As we can see, the results of automated deployment experiment 1 (A1) are comparable to the best handtuned results - slightly higher p_D and slightly higher p_F – a strong argument for automation. Additionally, by comparing A1 with A2, we observe that performance suffers as fewer resources are available. This is explained by the flow startup time - the time it takes since when the flow is requested by the learner to the time it starts pro-

Test	Nodes	Flow Startup Time	p_D	p_F
HandTuned	8 nodes	–	0.75	0.045
A1	8 nodes	90 sec	0.77	0.05
A2	4 nodes	205 sec	0.72	0.14

Table 2: Prediction Performance.

ducing predictions. In A2, with 4 nodes, most nodes are busy, hence it takes longer for a requested flow to be deployed (on average 205 seconds, during which time the flow misses processing around 240 beats)- and hence the prediction performance suffers. The difference of 0.1 in false alarm rate corresponds to around 12000 more normal beats being labeled ectopic.

We are conducting more detailed validation of these results with a more complex analytic flow space (with larger number of possible flows), with a dynamic change in the available analytics, and with finer grained performance and resource measurements.

4 Conclusion and Next Steps

We present a system that uses combinations of planning and machine learning to automate the orchestration of analytic workflows in Big Data settings. We use planning to identify feasible analytic workflows given descriptions of composition patterns and individual analytical building blocks. We use learning to explore the space of possible workflows and automatically identify appropriate combinations of these flows to deploy in response to dynamically changing data characteristics. We deploy this system to tackle a real-time ectopic beat detection problem in healthcare, and show that the automated system is able to produce results comparable with the best hand-tuned analytics. We are in the process of replicating these results across other domains such as cybersecurity, and using other Big Data platforms such as Hadoop. Interesting directions for future research include the use of hierarchical learning and planning, system resource scheduling and adaptation, and combining these with domain-specific reasoning to exploit domain expertise better.

References

[1] BLOUNT, M., EBLING, M., EKLUND, J., JAMES, A., MCGREGOR, C., PERCIVAL, N., SMITH, K., AND SOW, D. Real-time analysis for intensive care: Development and deployment of the artemis analytic system. *IEEE Engineering in Medicine and Biology Magazine 2* (2010), 110–8.

[2] BLUM, A., KALAI, A., AND LANGFORD, J. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory (COLT)* (1999), pp. 203–208.

[3] BULITKO, V., AND KOENIG, S. Planning and learning in a priori unknown or dynamic domains. In *Proceedings of the IJCAI 2005 Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains* (2005).

[4] CESA-BIANCHI, N., AND GENTILE, C. Improved risk tail bounds for on-line algorithms. In *NIPS* (2005).

[5] CESA-BIANCHI, N., SHALEV-SHWARTZ, S., AND SHAMIR, O. Efficient learning with partially observed attributes. *Journal of Machine Learning Research 12* (2011), 2857–2878.

[6] DUCHI, J., HAZAN, E., AND SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12* (2011), 2121–2159.

[7] DUDÍK, M., HSU, D., KALE, S., KARAMPATZIAKIS, N., LANGFORD, J., REYZIN, L., AND ZHANG, T. Efficient optimal learning for contextual bandits. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI)* (2011), pp. 169–178.

[8] FOX, M., AND LONG, D. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20*, 2003 (2003), 61–124.

[9] HAZAN, E., AND KOREN, T. Linear regression with limited observation. In *Proceedings of the 29th Conference on Machine Learning (ICML)* (2012).

[10] HILARIO, M., KALOUSIS, A., NGUYEN, P., AND WOZNICA, A. A data mining ontology for algorithm selection and meta-mining. In *ECML/PKDD Workshop on Third-Generation Data Mining: Towards Service-Oriented Knowledge Discovery (SoKD-09)* (2009).

[11] KARAMPATZIAKIS, N., AND LANGFORD, J. Online importance weight aware updates. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI)* (2011), pp. 392–399.

[12] MCMAHAN, H. B., AND STREETER, M. J. Adaptive bound optimization for online convex optimization. In *Proceedings of the 23rd Conference on Learning Theory (COLT)* (2010), pp. 244–256.

[13] MIT PhysioBank: Arrhythmia Database. `http://www.physionet.org/physiobank/database/mitdb/`.

[14] PEREZ, J., GERMAIN-RENAUD, C., KELEG, B., AND LOOMIS, C. Utility-based reinforcement learning for reactive grids. In *Proceedings of the Fifth International Conference on Autonomic Computing* (2008), pp. 205–6.

[15] RANGANATHAN, A., RIABOV, A., AND UDREA, O. Mashup-based information retrieval for domain experts. In *Proceedings of the 18th ACM conference on Information and knowledge management* (2009), ACM, pp. 711–720.

[16] RATLIFF, N., SILVER, D., AND BAGNELL, J. A. D. Learning to search: Functional gradient techniques for imitation learning. *Autonomous Robots 27* (2009), 25–53.

[17] RIABOV, A., AND LIU, Z. Scalable planning for distributed stream processing systems. In *Proceedings of ICAPS 2006* (2006).

[18] SIRIN, E., PARSIA, B., WU, D., HENDLER, J., AND NAU, D. HTN planning for Web service composition using SHOP2. *Journal of Web Semantics 1*, 4 (2005), 377–396.

[19] IBM Infosphere Streams. `http://www-01.ibm.com/software/data/infosphere/streams/`.

[20] TESAURO, G., JONG, N., DAS, R., AND BENNANI, M. Hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the Third International Conference on Autonomic Computing* (2006), pp. 65–73.

[21] Weka data mining in Java. `http://www.cs.waikato.ac.nz/ml/weka/`.

ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters

Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, and Johan de Kleer
Palo Alto Research Center, Palo Alto, CA, USA
{sgupta, cfritz, bprice, rhoover, dekleer}@parc.com

Cees Witteveen
Delft University of Technology, The Netherlands
c.witteveen@tudelft.nl

Abstract

Hadoop is the de-facto standard for big data analytics applications. Presently available schedulers for Hadoop clusters assign tasks to nodes without regard to the capability of the nodes. We propose *ThroughputScheduler*, which reduces the overall job completion time on a clusters of heterogeneous nodes by actively scheduling tasks on nodes based on optimally matching job requirements to node capabilities. Node capabilities are learned by running probe jobs on the cluster. ThroughputScheduler uses a Bayesian, active learning scheme to learn the resource requirements of jobs on-the-fly. An empirical evaluation on a set of sample problems demonstrates that ThroughputScheduler can reduce total job completion time by almost 20% compared to the Hadoop FairScheduler and 40% compared to FIFOScheduler. ThroughputScheduler also reduces average mapping time by 33% compared to either of these schedulers.

1 Introduction

Map-Reduce frameworks, such as Hadoop, are the technology of choice for implementing mayn big data applications. However, Hadoop and other frameworks typically assume a homogeneous cluster of server nodes and assign tasks to nodes regardless of their capabilities, while in practice, data centers may contain a heterogeneous mix of servers. When the jobs executing on the cluster also have heterogeneous resource requirements, which is typical, then it is possible to significantly increase processing throughput by actively matching jobs to server capabilities [2, 4, 6]. In this paper, we present the *ThroughputScheduler*, which actively exploits the heterogeneity of a cluster to reduce the overall execution time of a collection of concurrently executing jobs with distinct resource requirements. This is accomplished without *any* additional input from the user or the cluster administrator.

Optimal task allocation requires knowledge about both the resource requirements of jobs and the resource capabilities of servers, e.g., their relative CPU and disk I/O speeds. The ThroughputScheduler derives server capabilities by running "probe" jobs on the cluster nodes. These capabilities drift very slowly in practice and can be evaluated at infrequent intervals, e.g., at cluster set-up. In contrast, each new job has a-priori unknown resource requirements. We therefore present a learning scheme to learn job resource requirements on-the-fly.

The practicality of our solution relies on the structure of jobs in Hadoop. These jobs are subdivided into *tasks*, often numbering in the thousands, which are executed in parallel on different nodes. Mapping tasks belonging to different jobs can have very different resource requirements, while mapping tasks belonging to the same job are very similar. This is true for the large majority of practical mapping tasks, as Hadoop divides the data to be processed into evenly sized blocks. For a given job, we can therefore use online learning to learn a model of its resource requirements from a small number of mapping tasks in an *explore* phase, and then *exploit* this model to optimize the allocation of the remaining tasks. As we will show, this can result in a significant increase in throughput and never reduces throughput compared to Hadoop's baseline schedulers (FIFO and FairScheduler). We focus on minimizing the overall time to completion of mapping tasks, which is typically the primary driver of overall job completion time.

The next section reviews scheduling in Hadoop, followed by a discussion of related work. We then define a model of task completion time based on server capabilities and task requirements. We derive a Bayesian experimental design for learning the parameters of this model online, and present a real-time heuristic algorithm to optimally schedule tasks onto available cluster nodes using this model. Finally, we show empirically that ThroughputScheduler can reduce overall job execution time by up to 40% on a heterogeneous Hadoop cluster.

2 Hadoop Scheduler

In this section we briefly review the scheduler of Hadoop YARN [1]. YARN has a central entity called the resource manager. The resource manager has two primary modules: *Scheduler* and *ApplicationManager*. For every incoming job the ApplicationManager starts an *ApplicationMaster* on one of the slave nodes. The ApplicationMaster makes resource requests to the resource manager and is also responsible for monitoring the status of the job. Jobs are divided into *tasks* and for every task the scheduler assigns a *container* upon the request from the corresponding ApplicationMaster. A container specifies the node to run the task on and a fixed amount of resources (memory and CPU cores). YARN supports allocating containers based on the available resources (as of now just based on memory) on the nodes, but it has no mechanism to determine the actual resource requirements of a job.

To coordinate the allocation of resources for concurrent jobs, Hadoop provides three different schedulers: FIFO-, Fair- and CapacityScheduler. FairScheduler is the most popular scheduler among all because it enables fairness among concurrently executing jobs by giving them equal resources. All of Hadoop's schedulers are unaware of the actual resource profiles of jobs and the capabilities of node in the cluster and therefore often allocate resources sub-optimally.

3 Related Work

Recently, researchers have realized that the assumption of a homogeneous cluster is no longer true in many scenarios and have started to develop approaches that improve Hadoop's performance on heterogeneous clusters.

Speculative execution, a feature of Hadoop where a task that takes longer to finish than expected gets re-executed preemptively on a second node assuming the first may fail, can lead to degraded performance on heterogeneous clusters. This is because the scheduler's model of how long a task should take does not take the heterogeneous resources into account, leading to many instances of unnecessary speculative executions for tasks executing on slower nodes. The *LATE Scheduler* [9] improves speculative executing for heterogeneous clusters, to only speculatively execute tasks that will indeed finish late using the concept of *straggler* tasks [3]. However, the approach assumes that the hardware capabilities and the task resource profiles are already known rather than being discovered automatically.

The *Context Aware Scheduler for Hadoop* (CASH) [5] assigns tasks to the nodes that are most capable to satisfy the tasks' resource requirements. Similar to our approach CASH learns resource capabilities and resource requirements to enable efficient scheduling. However, unlike our online learning, CASH learns capabilities and requirements in offline mode. The performance of CASH is evaluated on a Hadoop simulator rather than a real cluster. Tian et al. propose a dynamic scheduler which learns job resource profile on the fly [8]. Their scheduler only considers the heterogeneity in the workload and assumes a homogeneous cluster to assign tasks to nodes. An architecture of a resource-aware cloud-driver for heterogeneous Hadoop clusters was proposed to improve the performance and increase fairness [7]. The cloud-driver tries to improve the performance by providing more efficient fairness among jobs in terms of resource allocation. Unlike our approach, the cloud-driver assumes that cluster capabilities are already known and it has abstract knowledge of job resource requirements.

4 Approach

In this section we describe the design for a scheduler that optimizes the assignment of tasks to servers. To do this, we need the *task requirements* and *server capabilities*. Unfortunately, these requirements and capabilities are not directly observable as there is no simple way of translating server hardware specifications and task program code into resource parameters. We take a learning based approach which starts with an *explore phase* where parameters are learned followed by an *exploit phase* in which the parameters are used to allocate tasks to servers. To learn these parameters by observation, we propose a task execution model that links observed execution times of map tasks to the unobservable parameters. We assume that map tasks belonging to the same job have very similar resource requirements. In the remainder of this section, we introduce the task model and then describe the explore and exploit phases.

4.1 Task Model

The task performance model predicts the execution time of a task on a server given the task resource requirements and the capabilities of the server node. We model a task as a set of resource specific operation types such as reading data from HDFS, performing computation, or transferring data over the network. The task resource requirements are represented by a vector $\theta = [\theta_1, \theta_2, \dots, \theta_N]$ where each component represents the total requirement for an operation type (e.g., number of instructions to process, bytes of I/O to read). The capabilities of the server are described by a corresponding vector $\kappa = [\kappa_1, \kappa_2, \dots, \kappa_N]$ which represent rates for processing the respective operation type (e.g., FLOPS or I/O per second).

In theory, some of these operations could take place simultaneously. For instance, some computation can occur while waiting for disk I/O. In practice this does not have a large impact on Hadoop tasks we studied. We therefore assume that the requirements for each operation type are processed independently. The time required to process a resource requirement is the total magnitude for the requirement divided by the processing rate. The total time T^j to process all resource requirements on server j is the sum of the times for each operation type

$$T^j = \sum_k \frac{\theta_k}{\kappa_k^j} + \Omega^j \tag{1}$$

where Ω^j is the overhead to start the task on the server. We assume that every job imposes the same amount of overhead on a given machine. In this paper, we consider a two dimensional model in which $\kappa = [\kappa_c, \kappa_d]$ represents computation and disk I/O server capabilities and $\theta = [\theta_c, \theta_d]$ represents the corresponding task requirements. Hence, the task duration model reduces to:

$$T^j = \frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j. \tag{2}$$

The parameters κ_c and κ_d abstractly capture many complex low-level hardware dependencies. For example, κ_c internally accounts for the kind of operations needed to be performed (flops or integer ops or memory ops). Similarly, κ_d is dependent on disk speed, seek time, etc. In practice, it is very difficult to build a task model as a function of these low level parameters. To keep the model simple and easier to understand we use such abstract parameters.

4.2 Explore

We learn server resource capabilities and task resource requirements separately. First we learn server capabilities offline. Then using these capabilities we actively learn the resource requirements for jobs online.

4.2.1 Learning Node Capabilities

We assume server capabilities κ^j's and overhead Ω^j do not change frequently and can be estimated offline. The server parameters are estimated by executing probe jobs. Since the time we measure is the only dimension with fixed units, the value of the parameters is underdetermined. We resolve the unidentifiability of the system by choosing a 'unit' map task to define a baseline. The unit map task has an empty map function and it does not read or write from/to HDFS.

The computation (θ_c) and disk task requirements (θ_d) are both zero, therefore Equation 2 allows us to estimate Ω. Multiple executions are averaged to create an accurate point estimate. Note that Ω includes some computation and disk I/O that occur during start up.

One could imagine attempting to isolate the remaining parameters in the same fashion, however, it is difficult to construct a job with zero computation or zero disk I/O. Instead we construct jobs with two different levels of resource usage defined by a fixed ratio η.

Let's assume we aim to determine κ_c. First we run a job $J_c^1 = \langle \theta_c, \varepsilon_d \rangle$ with fixed disk requirement ε_d (J_c^1 might be a job which simply reads an input file and processes the text in the file). We compute the average execution time of this job on each server node. According to our task model the average mapping time for every machine i can be given as

$$T_1^i = \frac{\theta_c}{\kappa_c^i} + \frac{\varepsilon_d}{\kappa_d^i} + \Omega^i \tag{3}$$

Next we run a job J_c^η which reads the same input but the processing is multiplied by η compared to J_c^1. Therefore, the resource requirements of J_c^η can be given as $J_c^\eta = \langle \eta\theta_c, \varepsilon_d \rangle$. The average mapping time for every node can be given as

$$T_n^i = \frac{\eta\theta_c}{\kappa_c^i} + \frac{\varepsilon_d}{\kappa_d^i} + \Omega^i \tag{4}$$

We solve for $\frac{\varepsilon_d}{\kappa_d}$ in equations 3 and 4, set them equal and solve for κ_c^i to get:

$$\kappa_c^i = \frac{\theta_c(\eta - 1)}{T_n^i - T_1^i} \tag{5}$$

This equation gives us κ_c^i in terms of a ratio. To make it absolute, we arbitrarily choose one node as the reference node. We set $\kappa_c^1 = 1$ and $\kappa_d^1 = 1$ and then solve equation 5 for θ_c. Once we have the task requirements θ_c in terms of the base units for server one, we can use this job requirement to solve for the server capabilities on all the other nodes. Similarly we estimate κ_d.

Normally in Hadoop, the output of map tasks goes to multiple reducers and may be replicated on several servers. This would have the effect of introducing network communication costs into the system. To avoid that while learning node capabilities, we set the number of reducers to zero and set the replication factor to one.

Table 1 gives an example of computed server capability parameters for a five node cluster of heterogenous machines. The algorithm correctly discovers that there are two classes of machines.

4.2.2 Learning Job Resource Profile

In this phase the resource requirements for tasks are learned in an online manner without interrupting production use of the cluster. To enable online learning we collect task completion time samples from actual production

Node	κ_c	κ_d	Ω
Node1	1	1	45
Node2	1	1	45
Node3	7.5	2.5	5.3
Node4	7.5	2.5	5.3
Node5	7.8	2.6	4.8

Table 1: Recorded Node Capabilities and Overhead

jobs. With every new time sample we update our belief about the resource profile $[\theta_c, \theta_d]$ of the job.

We assume that the observed execution time T^j is normally distributed around the value predicted by the task duration model given by Eq. 2. Given a distribution over resource parameters $[\theta_c, \theta_d]$, the remaining uncertainty due to changing conditions on the server (i.e., the observation noise) is given by a standard deviation σ_j.

$$T^j \sim \mathcal{N}\left(\frac{\theta_c}{\kappa_c^j} + \frac{\theta_d}{\kappa_d^j} + \Omega^j, \sigma^j\right) \tag{6}$$

Starting with prior beliefs about task requirements $p(\theta_c, \theta_d)$ and the execution model based likelihood function $p(T^j \mid \theta_c, \theta_d, \kappa_c^j, \kappa_d^j, \sigma^j)$, Bayes' rule allows us to compute a joint posterior belief over $[\theta_c, \theta_d]$:

$$p(\theta_c, \theta_d \mid T^j, \kappa_c^j, \kappa_d^j, \sigma^j) = \alpha p(T^j \mid \theta_c, \theta_d, \kappa^j, \sigma^j) p(\theta_c, \theta_d)$$

For our two-dimensional CPU and disk usage example, the likelihood has the form (Empirically we observed an observed variance of approximately +/- 3 indicates a standard deviation of 1, therefore, $\sigma^j = 1$):

$$p(T^j \mid \theta_c, \theta_d; \kappa_c, \kappa_d) = \frac{1}{\sqrt{2\pi}} \exp \frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j} - \Omega^j\right)^2}{2}$$

Note that the execution time is normally distributed around a line defined by the server capabilities $[\kappa_c, \kappa_d]$. The joint distribution of the likelihood is not a bivariate normal, but a univariate Gaussian tube around a line. This makes sense, as a given execution time could be due to a slow CPU and fast disk or a fast CPU and slow disk.

When a job is first submitted we assume that the resource requirements for its tasks are completely unknown. Assuming an uninformative prior, the posterior distribution after the first observation is just proportional to the likelihood.

$$p(\theta_c, \theta_d \mid T^j) = \frac{1}{\sqrt{2\pi}\sigma^j} \exp \frac{\left(T^j - \frac{\theta_c}{\kappa_c} - \frac{\theta_d}{\kappa_d} - \Omega^j\right)^2}{2}$$

For the second and subsequent updates we have a definite prior distribution and likelihood function. These two are multiplied to obtain the density of the second posterior update. Let the first experiment be on machine j with capability κ^j and let the observed time be T^j. Let the second experiment be on machine k with capability κ^k and let the observed time be T^k. The resulting posterior distribution is

$$p(\theta_c, \theta_d \mid T^j, T^k) = \frac{1}{\sqrt{2\pi}} \exp \left[\frac{\left(T^j - \frac{\theta_c}{\kappa_c^j} - \frac{\theta_d}{\kappa_d^j} - \Omega^j\right)^2}{2} + \frac{\left(T^k - \frac{\theta_c}{\kappa_c^k} - \frac{\theta_d}{\kappa_d^k} - \Omega^j\right)^2}{2} \right] \tag{7}$$

We omit the derivation for space, but we do give the update rules here. With every time sample we can recover the mean μ_{θ_c,θ_d} and covariance matrix $\Sigma_{\theta_c,\theta_d}$ by using the property of the bivariate Gaussian distribution. Expanding the exponent of Equation 7 and collecting the θ_c and θ_d term gives us a conic section in standard form:

$$a_{20}\theta_c{}^2 + a_{10}\theta_c + a_{11}\theta_c\theta_d + a_{01}\theta_d + a_{02}\theta_d{}^2 + a_{00} = 0 \tag{8}$$

There is a transformation to map between the coefficients of a conic in standard form and the parameters of a Gaussian distribution. The mean and covariance of the distribution with the same elliptical form is given by:

$$\begin{bmatrix} \mu_{\theta_c} \\ \mu_{\theta_d} \end{bmatrix} = \begin{bmatrix} (a_{11}a_{01} - 2a_{02}a_{10})/(4a_{20}a_{02} - a_{11}^2) \\ (a_{11}a_{10} - 2a_{20}a_{01})/(4a_{20}a_{02} - a_{11}^2) \end{bmatrix} \tag{9}$$

$$\Sigma^{-1}_{\theta_c\theta_d} = \begin{bmatrix} a_{20} & \frac{1}{2}a_{11} \\ \frac{1}{2}a_{11} & a_{02} \end{bmatrix} \tag{10}$$

For every new time sample we compute coefficients a_{nm} for equation 8. These coefficients determine the updated value of μ_{θ_c}, μ_{θ_d}, and $\Sigma_{\theta_c,\theta_c}$.

Because we recover both the mean and the covariance of task requirements, we can quantify our degree of uncertain about task requirements, and hence decide whether to keep exploring or starting to exploit this knowledge for optimized task scheduling. In this paper we sample tasks until we get a determinant for the covariance matrix $|\Sigma_{\theta_c,\theta_d}| < 0.007$. Table 2 summarizes resource requirements learned by the online inference mechanism for some of the Hadoop example jobs. When we compare the 'Pi' job, which calculates digits of Pi, to RandomWriter, which writes bulk data, we see that the algorithm correctly recovers the fact that Pi is compute intensive (large μ_{θ_c}) whereas RandomWrite is disk intensive (large μ_{θ_d}). Other Hadoop jobs show intermediate resource profiles as expected. The J_{IO} job will be described further in the experimental section. The '# of Tasks' column gives the number of tasks executed to reach the desired confidence.

4.3 Exploit

Once the resource profile of a job is learned to sufficient accuracy we switch from explore to exploit. The native Hadoop scheduler sorts task/machine pairs according to

Job	μ_{θ_c}	μ_{θ_d}	$\lvert\Sigma_{\theta_c\theta_d}\rvert$	# of Tasks
Pi	24.00	6.30	0.0038	109
Random Writer	27.26	234.62	0.0061	28
Grep	15.82	8.10	0.0038	90
WordCount (1.5 GB)	43.50	22.50	0.00614	31
WordCount (15 GB)	138.05	206.40	0.00615	32
J_{IO}	5.60	96.46	0.0063	30

Table 2: Job resource profile measurements with variance and number of tasks executed

whether they are local (data for the task is available on the machine), on the same rack, or remote. We introduce our routine based on our task requirements estimation called "SelectBestJob" to break ties within each of these tiers as shown in Algorithm 4.1: If we have two local jobs, we would run the one most compatible with the machine first.

Algorithm 4.1: THROUGHPUTSCHEDULER(Cluster, Request)

```
for each Node N $\in$ Cluster
do
    JobsWithLocalTasks $\leftarrow$ N.GETJOBSLOCAL(Request)
    JobsWithRackTasks $\leftarrow$ N.GETJOBSRACK(Request)
    JobsWithOffSwitchTasks $\leftarrow$ N.GETJOBSOFFSWITCH(Request)
    if LocalJobs $\neq NULL$
      then
        $J \leftarrow$ SELECTBESTJOB(LocalJobs, N)
        ASSIGNTASKFORJOB(N, J)
    else if RackJobs $\neq NULL$
      then
        $J \leftarrow$ SELECTBESTJOB(RackJobs, N)
        ASSIGNTASKFORJOB(N, J)
    else
        $J \leftarrow$ SELECTBESTJOB(OffSwitchJobs, N)
        ASSIGNTASKFORJOB(N, J)
```

Algorithm 4.2: SELECTBESTJOB(*NodeN, ListofJobs*)

```
return $(\text{argmin}_{J \in \text{ListOfJobs}} \frac{\text{norm}(\theta_c^J)}{\text{norm}(\kappa_c^N)} + \frac{\text{norm}(\theta_c^J)}{\text{norm}(\kappa_c^N)})$
```

SelectBestJob, shown in Algorithm 4.2, selects job J that minimizes a *score* for task completion on node N. However, rather than using absolute values of θ_c, θ_d, κ_c and κ_d, we use the normalized value of these parameters to define the score. While absolute values represent expected time of completion, which can be measured in seconds, job selection based on these numbers would always favor short tasks over longer once and fast machines over slower ones. This would not achieve the optimized matching of job requirements to server capabilities. For example, consider Nodes 1 and 3 in Table 1. Node 3 is almost 7.5 times faster than Node 1 in terms of CPU, but only 2.5 times faster in terms of disk. Hence, intuitively, disk intense jobs are better scheduled on Node 1, since the relativly higher CPU performance of Node 3 is better used for CPU intense jobs (if there are any). To account for this relativity of optimal resource matching, we normalize both jobs and machines to make their total requirements and capabilities sum to one for each resource x (here $x \in \{c, d\}$):

$$\text{norm}(\theta_x^i) = \frac{\mu_{\theta_x^i}}{\sum_k \mu_{\theta_k^i}} \qquad \text{norm}(\kappa_x^j) = \frac{\kappa_x^j}{\sum_{k=1}^5 \kappa_x^k}$$

5 Experimental Results

To evaluate the performance of *ThroughputScheduler* we conducted experiments on a five node Hadoop cluster at PARC (see Table 1).

5.1 Evaluation on Heterogeneous Jobs

We evaluate the performance of our scheduler on jobs with different resource requirements. Since the Hadoop benchmarks do not contain highly I/O intensive jobs (cf. Table 2), we constructed our own I/O intensive Map-Reduce job, J_{IO}. J_{IO} reads 1.5 GB from HDFS, and writes files totaling 15 GB back to HDFS. This resembles the resource requirements of many expand-translate-load (ETL) applications used in big data applications to pre-process data using Map-Reduce and writing into HBase, MongoDB, or another disk-backed database. We learn J_{IO}'s resource profile using the job learner described in the Explore section. The learned resource requirement of J_{IO} is listed in Table 2. To evaluate ThroughputScheduler on drastically heterogeneous job profiles, we run J_{IO} along with the Hadoop benchmark *Pi*, which is CPU intense. We compare the performance of ThroughputScheduler with FIFO- and FairScheduler—for a single user, CapacityScheduler is no different from FIFO.

5.1.1 Job Completion Time

We first compare the performance of the proposed scheduler in terms of overall job completion time. In case of multiple jobs, the overall job completion time is defined as the completion time of the job finishing last. In this experiment we study the effect of heterogeneity between job resource requirements, which we can quantify as the ratio of disk I/O to CPU requirement of a job: $h = \frac{\theta_d}{\theta_c}$. In order to vary this quantity we vary the I/O load of J_{IO} further by varying the replication factor of the cluster: the higher the replication factor, the higher the I/O load of a job. This impacts disk I/O intense jobs more than others.

These results show that ThroughputScheduler performs better than FIFO- and FairScheduler in all cases. The relative performance increase of our scheduler increases as the heterogeneity of the two jobs increase, as simulated by an increased replication factor: up to 40% compared to FIFO, and 20% compared to Fair. Note that both the Fair- and the ThroughputScheduler benefit from

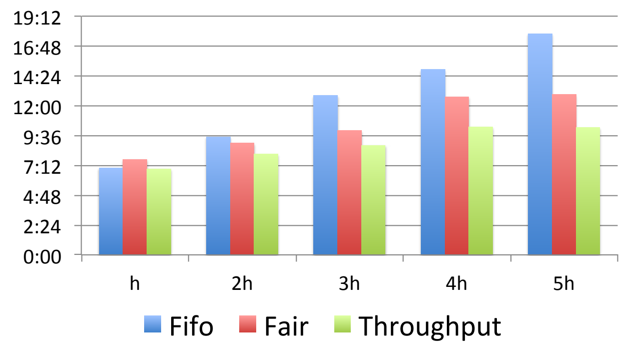


Figure 1: Overall job completion time in minutes (Y axis) on heterogeneous nodes at PARC for different relative values of $h = \frac{\theta_d}{\theta_c}$. Disk load θ_d is increased by increasing the replication number.

higher replication as they can better take advantage of data locality. The improvements of ThroughputScheduler beyond Fair- are purely due to our improved matching of jobs to computational resources.

Job	FIFO	Fair	Throughput
Pi	9 sec	9 sec	6 sec
J_{IO}	2 min 15 sec	2 min	2 min 10 sec

Table 3: Comparison of Average Mapping Time

To better understand the source of this speed-up, we considered the average mapping time for each job (*throughput*). Table 3 summarizes these results and provides the explanation for the speed-up: our scheduler improves the throughput of *Pi* by 33%, while maintaining the throughput of J_{IO} compared to the other schedulers. Since *Pi* has very many mapping tasks, these savings pay off for the overall time to completion.

5.2 Performance on Benchmark Jobs

To estimate the performance of ThroughputScheduler on realistic workloads, we also experimented with the existing Hadoop example jobs. We ran the job combinations of concurrent jobs shown in Table 4.

$Comb_1$	Grep (15 GB) + Pi (1500 samples)
$Comb_2$	WordCount (15 GB) + Pi (1500 samples)
$Comb_3$	WordCount (15 GB) + Grep (15 GB)

Table 4: Job Combination

The performance comparison in terms of job completion time is presented in Figure 2. For these workloads ThroughputScheduler performs better than either of the other two in all cases. For $Comb_2$ the job completion time is reduced by 30% compared to FIFO. For $Comb_3$ all three schedulers perform similarly because both jobs are CPU intensive (cf. Table 2).

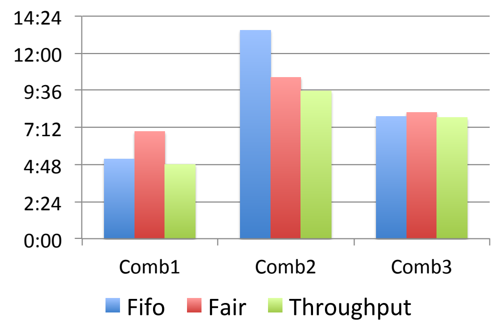


Figure 2: Job Completion time in minutes (Y axis) of combinations of Hadoop example jobs.

Job Combination	FIFO	Fair	Throughput
$Pi(1500sample), WC(15GB)$	440s	319s	310s
$Pi(1500sample), Grep(15GB)$	210s	224s	214s
$WC(15GB), Grep(15GB)$	225s	262s	214s

Table 5: Completion time of job combinations on a homogeneous cluster.

5.3 Performance on Homogeneous Cluster

We ran additional experiments on a set of homogeneous cluster nodes, to ensure such a setup would not cause ThroughputScheduler to produce inferior performance. These results are shown in Table 5.

6 Conclusion

ThroughputScheduler represents a unique method of scheduling jobs on heterogeneous Hadoop clusters using active learning. The framework learns both server capabilities and job task parameters autonomously. The resulting model can be used to optimize allocation of tasks to servers and thereby reduce overall execution time (and power consumption). Initial results confirm that ThroughputScheduler performs better than the default Hadoop schedulers for heterogenous clusters, and does not negatively impact performance even on homogeneous clusters.

While our demonstration uses the Hadoop system, the approach implemented by ThroughputScheduler is applicable to other framework of distributed computing as well.

References

[1] Apache hadoop nextgen mapreduce (yarn). `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[2] Balakrishnan, S., Rajwar, R., Upton, M., and Lai, K. The impact of performance asymmetry in emerging multicore architectures. In *In Proceedings of the 32nd Annual International Symposium on Computer Architecture* (2005), pp. 506–517.

[3] Bortnikov, E., Frank, A., Hillel, E., and Rao, S. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, pp. 18–18.

[4] Ghiasi, S., Keller, T., and Rawson, F. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd conference on Computing frontiers* (New York, NY, USA, 2005), CF '05, ACM, pp. 199–210.

[5] Kumar, K. A., Konishetty, V. K., Voruganti, K., and Rao, G. V. P. Cash: context aware scheduler for hadoop. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics* (New York, NY, USA, 2012), ICACCI '12, ACM, pp. 52–61.

[6] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. Heterogeneous chip multiprocessors. *Computer 38*, 11 (Nov. 2005), 32–38.

[7] Lee, G., Chun, B.-G., and Katz, H. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2011), HotCloud'11, USENIX Association, pp. 4–4.

[8] Tian, C., Zhou, H., He, Y., and Zha, L. A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on* (2009), pp. 218–224.

[9] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 29–42.

Real-Time User-Centric Management of Time-Intensive Analytics Using Convergence of Local Functions

Invited position paper

Vinay Deolalikar
HP-Autonomy Research
`vinay.deolalikar@hp.com`

Abstract

The past decade has witnessed an astonishing growth in unstructured information in enterprises. The commercial value locked in enterprise unstructured information is being increasingly recognized. Accordingly, a range of textual document analytics—clustering, classification, taxonomy generation, provenance, etc.— have taken center stage as a potential means to manage this explosive growth in unstructured enterprise information, and unlock its value.

Several analytics are time-intensive: the time taken to complete processing the increasingly large volumes of data is significantly more than real-time. However, users are increasingly demanding *real-time* services that rely on such time-intensive analytics. There is clearly a tension between the aforementioned two developments.

In light of the preceding, vendors increasingly realize that *while an analytic may take a longer time to converge, they need to extract useful information from it in real-time. Furthermore, this information has to be application-driven.* In other words, it is often not an option to simply "wait until the analytic has finished running:" they must start providing the user with information while the analytic is still running. In summary, there is an emerging stress in Enterprise Information Management (EIM) on application-driven real-time information being extracted from time-intensive analytics.

A priori, it is not clear what could be extracted from an analytic that has yet to complete, and whether any such information would be useful. As of the present, there is little or no research literature on this problem: it is generally assumed that all of the information from an analytic will be available upon its completion.

We present an approach to this problem that is based on decomposing the objective function of the analytic, which is a global function that determines the progress of the analytic, into *multiple local, user-centric functions.* How can we construct meaningful local functions? How can such functions be measured? How do these functions evolve with time? Do these functions encode useful information that can be obtained real-time? These are the questions we will address in this paper.

We demonstrate our approach using local functions on document clustering using the de facto standard algorithm—*k*-means. In this case, the multiple local user-centric functions transform *k*-means into a flow algorithm, with each local function measuring a flow. Our results show that these flows evolve very differently from the global objective function, and in particular, may often converge quickly at many local sites. Using this property, we are able to extract useful information considerably earlier than the time taken by *k*-means to converge to its final state.

We believe that such pragmatic approaches will have to be taken in order to manage systems performing analytics on large volumes of unstructured data.

1 Introduction

1.1 Enterprise Information Management

Enterprises spend billions of dollars annually to manage *unstructured information*; namely information that exists mostly as text in documents having multiple formats, but no fixed schema (unlike, say, a database which is queried using SQL). These documents reside on desktops, laptops, email exchanges, web and file servers, wikis, and sharepoint repositories. This segment of enterprise information is growing much faster than structured information, and already it is estimated that 70% of all information in an enterprise exists in unstructured formats.

Due to the lack of structure, managing unstructured information poses unique challenges. Currently, major drivers for these management efforts include applications such as eDiscovery, compliance requirements for different categories of documents necessitated by new laws such as HIPAA, IT management operations, document searches made by employees in various capacities, sales force support needs, and a host of other applications. Analytics[1] of various types—clustering, classification, taxonomy extraction, to name a few prominent ones—are generally regarded as the primary techniques that will help meet these challenges and enable such applications. Analytics for enterprise unstructured information, viewed through the prism of end-user applications, form the broad context for our work.

1.2 Emerging Problem: Real-Time Extraction of Information

Analytics, generally speaking, uncover relationships in unstructured information. The greater the volume of the information, the more time it takes for an analytic to process the information, and extract relationships. Workflows in enterprise information management are increasingly complex, and require analytics inputs at various stages. These stages are pipelined together. Users of these applications demand the ability to perform workflows in near real-time: any application that requires the user to "wait until the current stage completes" is a significant dent on market acceptance.

Therefore, on the one hand, analytics are needed to enable applications that require unstructured information management at scale. On the other hand, the time taken by a particular analytic to complete at scale prevents its use in the application.

In light of the above catch-22 situation, there has been a great deal of attention devoted to making analytics run faster. We wish, instead, to highlight an emerging paradigm: vendors are increasingly trying to obtain "just enough" information from an analytic that can satisfy the current need of the user, and enable them to move to the next stage of their workflow. In other words, they want to provide enough information to the user that hides the actual run time of the analytic from them. The analytic may well take significantly longer to complete, but how can we extract useful application-enabling information from it in near real-time? As of now, there is little or no work on this question, and to our knowledge we are the first to frame it explicitly. We believe that this question will become increasingly important as the scales of unstructured information grow.

1.3 Our Approach: Local User-Centric Functions

Most analytics try to optimize (usually minimize or maximize) some global *objective function*. For example, clustering tries to minimize the sum of distances of documents to cluster centroids. However, these objective functions are mathematical objects: end-users do not usually think in terms of objective functions. Rather, they have more application-centric concerns.

Our approach is to try to "partition" the global objective function into local functions that are user and application-centric, and that capture what the user might be interested in from the analytic. Then, we will try to measure these local functions. The hope is that while the global objective function captures the overall convergence behavior of the analytic, these local functions might already start yielding information to the user that is precisely of the form that they are interested in. The idea is that while the global state of the analytic is determined by the global objective function, the evolution of local states might be tracked using our local functions. These local functions, if they are appropriately constructed, might give users information that they can start acting upon immediately. Furthermore, by viewing the analytic as a conglomeration of locally evolving states, we can provide information to the user "piecemeal" instead of all at once, especially since that more accurately reflects how the user will digest the information anyway.

[1]An analytic is, broadly, a functionality that examines data, analyzes it, and draws inference based upon the results of the analysis.

1.4 Contributions

Our main contributions are sketched below:

1. We frame the question of real-time piece-by-piece extraction of information from time-intensive analytics. We believe that this question will be increasingly important in the future, given the explosive growth of unstructured information.

2. We present a novel approach to the problem above, based on inspecting analytics algorithms locally using user-defined functions.

3. We work out our approach for an important analytic—text clustering—that is key to several EIM applications.

2 Key Idea: Defining User-Centric Objectives with Local Functions

As stated in the introduction, most analytics are defined in terms of an objective function that is to be maximized/minimized over the course of the run-time of the analytic. For example, k-means clustering is often defined as follows. Given m data-points $x = \{x_1, \ldots, x_m\}$, each of which is a d-dimensional vector, find k "means" $\mu = \{\mu_1, \ldots, \mu_k\}$, also d-dimensional, such that the following objective function is minimized.

$$E(x,\mu) = \sum_{i=1}^{m} L(x_i, \mu_j), \tag{1}$$

where μ_j is the closest of the k means to x_i in terms of the norm L.

We may write the function above as a sum of sums: each outer sum would pertain to a single μ_j. Thus,

$$E(x,\mu) = \sum_{i=1}^{m} \sum_{j=1}^{k} L(x_i,\mu_j), \tag{2}$$

where the inner sum is over all data-points that are closest to μ_j. The general form of the objective function then becomes "optimize some global function (the sum, in the case above) of local pieces." A "local piece" here is the inner sum that pertains only to a single cluster, and therefore *can be computed locally at each cluster*.

Moreover, as noted earlier, objective functions such as (1) are far from the mind of the end-user of an analytic. The user is concerned with something that *describes the problem from their perspective*. The enterprise user frequently wants to associate some meaning to the information that the analytic extracts as it runs.

Can we then, partition the objective function into local pieces, with the additional desideratum of making each local piece pertain to the user's requirements? How do these local functions converge? Do they all converge uniformly, at the same rate as the global function, or do they display non-uniform convergence behavior? Do a majority of them converge quickly, well before the convergence of the global objective function? These are some of the questions our empirical work will try to uncover.

At this time, we will empirically analyze, in some detail, our chosen example analytic—document clustering with k-means. We choose document clustering with k-means because it is arguably the first analytic that a user might want to run in a large number of enterprise applications. Most EIM vendors today offer the ability to cluster a user's data, but several applications which could potentially use this clustering do not do so since it takes considerably longer than real-time to finish clustering a large dataset. In summary, we are aware of several applications that need clustering, but currently rely on a static, older clustering of the data instead of allowing the user to dynamically cluster data as they proceed through their workflow. We show how our approach can mitigate this situation, and how we can instrument k-means with local user-centric functions to extract near real-time information that is useful to the user at their current stage of workflow.

3 Example: Document Clustering with k-means

The k-means algorithm is ubiquitous in data mining [10]. k-means can be used at various stages in EIM: to understand high-level organization of data [2, 5, 6], to organize search results [3], to extract semantic information such as labels [4], and so on. k-means is also time-intensive, and therefore a good candidate for us to demonstrate our approach.

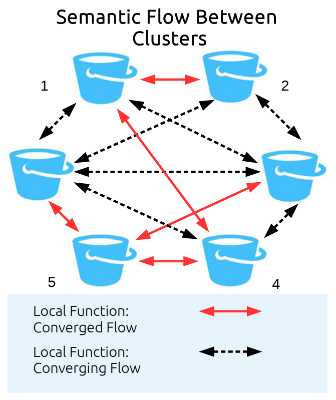

Semantic Flow Between Clusters
1
2
3
4
5
6
Local Function: Converged Flow
Local Function: Converging Flow

Figure 1: Schematic of the view of k-means through local functions: k-means is a set of pairwise flows, most of which abate early. Once flows to and from a cluster have abated—as has happened to Cluster 5—, we may extract semantic meaning from it. This can happen very early during run-time, long before final convergence of k-means.

3.1 Preliminaries

We briefly provide the framework of the document representation we use. Since we are clustering text, we use a tf-idf weighted bag-of-words vector representation for the documents [7]. We use a standard stoplist, and remove all words that occur fewer than three times in the corpus. As is standard, we normalize each vector to unit length so that if two documents of different lengths are still speaking of the same topics, they are regarded equally [8]. Finally, we use the cosine of the angle between the vectors as our similarity metric since it is known to outperform metrics such as Euclidean distance for text applications [9].

We use the random assignment version of k-means (as opposed to Forgy), where each document is randomly assigned a cluster at initialization. Cluster centroids are then computed, and re-assignment of documents to the closest (in terms of cosine similarity) cluster is done iteratively until there is no further movement of documents between clusters.

3.2 User-Centric Objectives: Concept Classes

A key component of our approach is to replace the focus on the global objective function with local user-centric functions. These functions should capture the domain-specific requirements of the user. What would such requirements be in the case of enterprise applications of document clustering?

A large majority of EIM applications that (could) use clustering want to understand the semantics, or "meaning" of each cluster. In other words, clustering is seen as a technique that groups the data into conceptually coherent groups, each group speaking of a coherent class of concepts. These concepts are then used in the next stage of the EIM pipeline: for example, they may be used in scatter-gather type workflows (for example, some eDiscovery workflows), to create taxonomies (for records management or classification), and so on. Therefore, the first task in capturing the user's requirements is to compute, from each cluster, a set of coherent concepts that it speaks about.

3.2.1 Cluster Digests: Concept Labels

Definition 1 *Define D_i as the Boolean indicator variable for document inclusion into C_i. Define $t_{j,D}$ as the Boolean indicator variable for term inclusion in a document.*

$$D_i[D,C_i] = \begin{cases} 1, & \textit{if } D \in C_i, \\ 0, & \textit{else;} \end{cases} \quad t_{j,D}[t_j,D] = \begin{cases} 1, & \textit{if } t_j \in D, \\ 0, & \textit{else.} \end{cases} \tag{3}$$

Therefore, D_i is a random variable whose arguments are $[D,C_i]$. $t_{j,D}$ is a random variable with arguments $[t_j,D]$. Therefore, for a fixed i and j, both of these are random variables over the set of documents. In this case, their mutual information is well defined.

Definition 2 *We define $I[i,j]$ as the mutual information between the random variables D_i and $t_{j,D}$.*

Conceptually, this measures the increase in (conditional) probability of a document being placed by the k-means algorithm in cluster C_i given that it has the term t_j. In practice, we also perform thresholding: namely, we only count those terms that occur at least five times in the corpus in order to preclude terms that may occur only in a few documents, all of whom land in one cluster.

Definition 3 *For $\ell > 0$, the ℓ* concept labels *associated to cluster C_i are the top ℓ terms $\{t_j\}$ in the corpus in descending order of $I[i,j]$. We denote the set of concept labels for C_i by T_i.*

3.3 Local Functions: Concept Flows

In order to demonstrate our approach, we construct certain functions that can be measured retroactively: namely, measuring them requires the algorithm to have converged. However, the empirical properties of these functions will suggest that, indeed, these functions can be approximated in real-time.

At each iteration of the k-means algorithm, documents move between clusters. We wish to measure how much information that is core to the cluster enters and leaves each cluster as a result of this.

In order to measure this "concept flow" when a document moves between cluster C_{i_1} and C_{i_2}, we measure the presence of terms in the document that are concept labels for C_{i_1} and C_{i_2}. By taking the difference of these two quantities, we obtain a measure of the "concept flow" associated to the movement of the document.

We wish to measure the flow of concepts in both directions between a pair of clusters, at any iteration.

Definition 4 *Let $\ell > 0$ and $m < n$. Let document D move from C_{i_1} and C_{i_2} at iteration m of k-means. Let $n_{j,D}$ be the number of times term t_j occurs in document D. The* forward concept flow *associated with the document D at iteration m is defined as*

$$\sigma_f[D,m] := \sum_{t_j \in T_{i_1}} n_{j,D}. \tag{4}$$

The reverse concept flow *associated with the document D at iteration m is defined as*

$$\sigma_r[D,m] := \sum_{t_j \in T_{i_2}} n_{j,D}. \tag{5}$$

The total concept flow *associated with the document D at iteration m is defined as*

$$\sigma_t[D,m] := \sigma_f[D,m] - \sigma_r[D,m]. \tag{6}$$

We also define the net quantities obtained by summing the above over all documents that move from one cluster to another.

Definition 5 *The* net forward concept flow *from cluster C_{i_1} to cluster C_{i_2} at iteration m is defined as*

$$\Sigma_f[i_1,i_2,m] := \sum_{D \in C_{i_1}} \sigma_f[D,m]. \tag{7}$$

The net reverse concept flow *from cluster C_{i_1} to cluster C_{i_2} at iteration m is defined as*

$$\Sigma_r[i_1,i_2,m] := \sum_{D \in C_{i_1}} \sigma_r[D,m]. \tag{8}$$

The net concept flow *from cluster C_{i_1} to cluster C_{i_2} at iteration m is defined as*

$$\Sigma_t[i_1,i_2,m] := \sum_{D \in C_{i_1}} \sigma_t[D,m]. \tag{9}$$

Finally, we define the average per-cluster-pair quantities.

Definition 6 *The* average forward concept flow *between and ordered cluster pair at iteration m is defined as*

$$\overline{\Sigma}_f[m] = \frac{1}{k(k-1)} \sum_{i_1,i_2;i_1 \neq i_2} \Sigma_f[i_1,i_2,m]. \tag{10}$$

The average reverse concept flow *between an ordered cluster pair at iteration m is defined as*

$$\overline{\Sigma}_r[m] = \frac{1}{k(k-1)} \sum_{i_1,i_2;i_1 \neq i_2} \Sigma_r[i_1,i_2,m]. \tag{11}$$

The average (net) concept flow *between an ordered cluster pair cluster at iteration m is defined as*

$$\overline{\Sigma}_t[m] = \frac{1}{k(k-1)} \sum_{i_1,i_2;i_1 \neq i_2} \Sigma_t[i_1,i_2,m]. \tag{12}$$

Notice that although we are measuring the semantic flow, as described above, during pre-convergence iterations of k-means, we obtain the labels only after convergence. Let c denote the iteration at which k-means converges. Then, measurement of semantic flows, with respect to the final labels, at iteration $m(< c)$ requires us to wait until convergence at iteration c. However, let us now inspect these flows carefully, and see if we may approximate them in real-time.

4 Experimental Results

4.1 Datasets and Protocol

We used two standard benchmark datasets for document clustering. The first is N20, the 20 Newsgroups dataset that contains roughly 20,000 articles posted to 20 usenet group. The articles are more or less evenly divided between the newsgroups; however some newsgroups are highly related, while others are not. The second dataset is REU, the Reuters-21758 dataset that has documents from Reuters newswire having 82 primary topics. For both datasets, we ran k-means with the "natural" number of clusters k—namely, 20 for N20, and 82 for REU.

We ran each clustering experiment five times. Since our results require us to examine concept flows between specific pairs of clusters, and these pairs change from experiment to experiment, we picked the experiment that was most typical of the five (in terms of convergence behavior) to depict our results. The variance between experiments was minor, and the form of the results did not change from experiment to experiment.

For the most typical experiment (as described above), the clustering of N20 took 35 iterations, while that for REU took 52 iterations. For each experiment, we ordered the $k(k-1)$ ordered pairs of clusters by descending order of semantic flows, summed over iterations [10, 15]. Next, we measured the changes in semantic flow for all these pairs as the experiment progressed. Fig. 2 shows results for REU.

4.2 Properties of Concept Flows

The inspection of the graphs in Fig. 2, and the similar graphs for N20 (which we could not show due to lack of space) immediately lead us to the following empirical result:

1. Local functions, unlike objective functions, are not monotone. The sequence $\Sigma_t[i_1,i_2,1], \Sigma_t[i_1,i_2,2], \ldots$ shows a zig-zag behavior until it falls to zero.
2. The average flow first rises sharply, but then starts to fall sharply after only a few (less than 5) iterations for both datasets. Compare this to the convergence time for each dataset (52 and 35 iterations, respectively).
3. The average reverse flow has also nearly abated by this time (i.e., by 5 iterations).
4. In the few cases of cluster pairs where flows are significant even after they have abated in other pairs; we found that the clusters themselves are semantically related.

These empirical results, repeated over multiple experiments, suggest that for a large majority of clusters, the "documents that matter" have already been placed into their correct clusters well before final convergence of k-means. Thus, our flow measurements uncover an "almost everywhere convergence" of k-means well before it converges globally in terms of its objective function.

We have experimented with other values for k, and the results are similar.

4.3 Near Real-Time Information

At this time, we can answer the question "what information can be extracted in near real-time as a result of the properties in §4.2?"

We have seen, empirically, that local flow functions for a majority of cluster pairs abate very quickly—between 5 and 10 iterations. At this time, we can extract concept classes for each cluster. For a majority of the clusters, these concept classes will continue to be accurate at convergence. The few clusters where these concept classes change significantly can be detected by our local concept flow function measurements, and updated accordingly. In this manner, we can already provide the user with a large proportion of the information that they desired from the analytic, but well before the analytic actually converges. In cases where k-means takes of the order of a few minutes to complete, the time taken to provide this information will be of the order of (tens of) seconds, which can enable a near real-time workflow.

The key idea behind our approach is to use the "almost everywhere convergence" to start providing local information to the user at places where such convergence has already happened, and not wait for global convergence.

In general, in any workflow where each cluster has to be further examined, we can supply the user with information on all the clusters that have already converged, so that they can begin examining those. This yields several examples of enterprise workflows where local information gathered as described above can enable real-time workflows. One example is a large eDiscovery workflow.

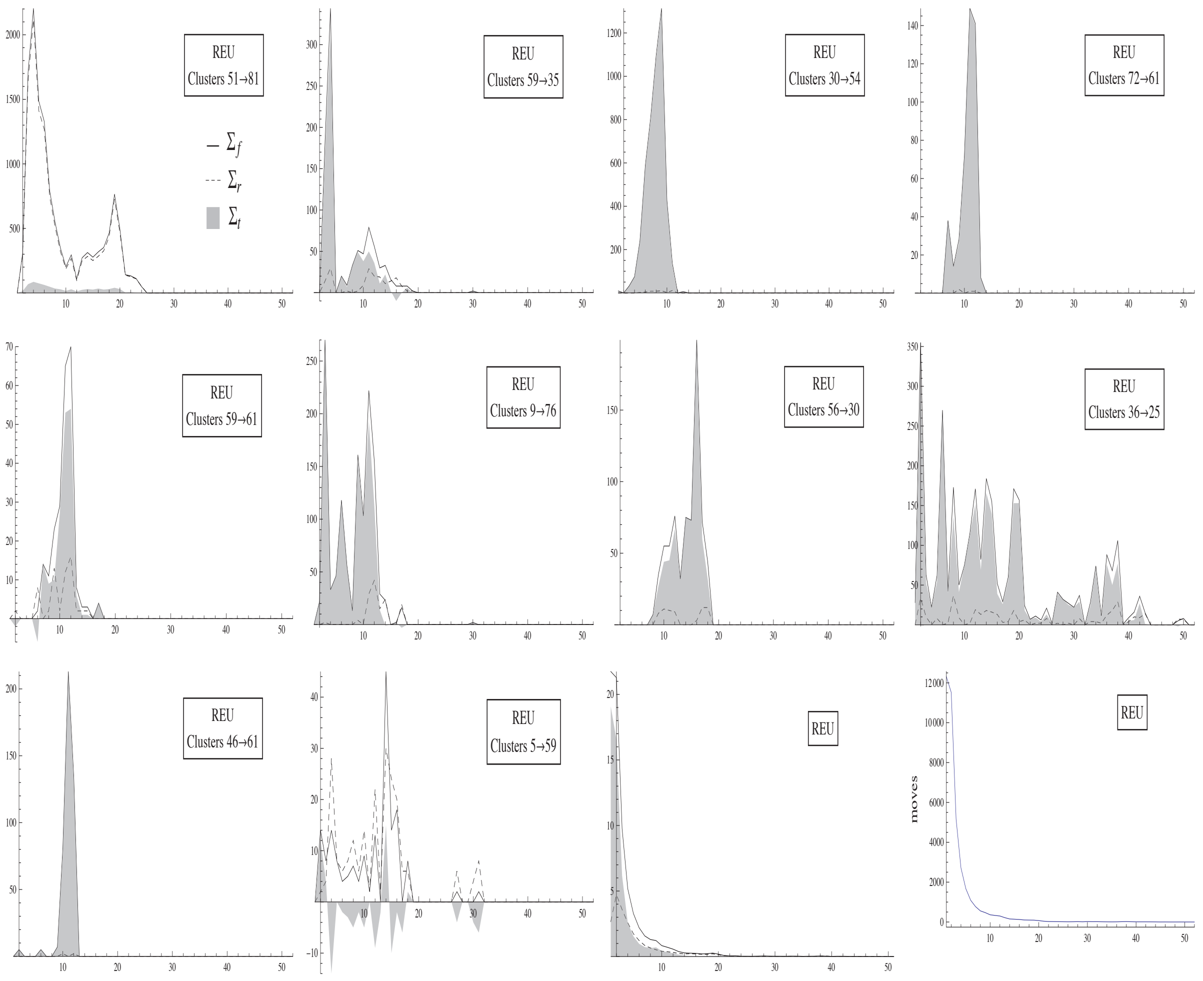


Figure 2: The first ten graphs show the top ten semantic flows for a run of *k*-means on REU. The Y-axis measures flows. A majority of the lower-ranked flows abate much quicker, and even the highest ranked flows tend to abate well before convergence. Legends are shown only on the first graph. The next graph is the average flow, taken over all pairs (not just the topmost). The final graph is the total number of documents that move at each iteration of *k*-means. In all graphs, the X-axis is the iteration number. For lack of space, only the experiments on REU are shown; similar results were observed on N20.

Mid-sized eDiscovery cases frequently have to examine a few hundred thousands of documents in a limited time-frame. For example, in early case assessment, this time frame might be only a few weeks. If clustering is used to organize the inspection of these documents, then the inspection of clusters that have already converged can begin as soon as their information is available, without waiting for clustering to converge throughout the corpus. The larger the corpus of documents, the more time is saved using this real-time enabled workflow, over a workflow where the user waits for corpus-wide convergence.

In any scatter-gather workflow [2], the user examines each cluster individually during the gather phase, and decides whether it should be included in the subsequent scatter phase. This represents another generic workflow where providing clusters as soon as they have converged can enable the use to make their decision on the available clusters, without waiting for the remaining clusters.

A natural question that may be asked is: can a cluster where convergence seems to have happened “change” its convergent state? Can it start showing an increased flow after having seemingly converged? First of all, we must ensure that the flow has indeed abated for a period of a few successive (say, 5) iterations. We did not observe significant flows after a abatement of flows lasting five iterations. Rarely, we do see a small additional flow in such cases, for example, the flow $5 \rightarrow 59$ shown in Fig. 2, but as is the case in the example, it is not very large.

What about pairs of clusters where flow has not abated till a relatively late stage? We can simply flag such pairs of clusters as being “semantically related,” to be considered together for scatter-gather. This accurately reflects, for instance, eDiscovery workflows.

4.4 Time to Measure Local Functions

With the following pragmatic choices, we can instrument k-means for real-time local flow functions at insignificant additional cost.

1. In order to measure the local functions $\Sigma_t[i_1, i_2, 1]$, we need $k(k-1)$ counters, one for each bidirectional measurement of flow between each pair of clusters. The time taken to updating each counter is dominated by the time to compute the actual move to be made for each document, and does not significantly increase their sum.

2. Moreover, this need not be done at every iteration, since all we want is to detect abatement of flows. We experimented with computing flow only at iteration 5, 10, 20, 30, and so on, yielding satisfactory results.

3. The time taken to compute labels can be significant; however, in light of the quick abatement of flows, we can compute these labels only once, soon after iteration 5.

5 Related Work

We are not aware of any work that studies the behavior of k-means with respect to local user-centric functions. However, more generally, our work may be seen as a study of the k-means algorithm during its convergence. In this regard, the work that is closest to ours is [1]. However, there are obvious and fundamental differences: besides the core difference of local vs. global functions, [1] studied the convergence of k-means on the IRIS dataset, which has only four dimensions. One of the primary properties of text document corpora that distinguish it is the high-dimensional and sparse nature of the feature vectors. As expected given these important differences, the results of the experiments (namely, the trajectories of the functions under study) vary greatly. As but one example, the behavior of the sequence $\Sigma_t[i_1, i_2, 1], \Sigma_t[i_1, i_2, 2], \ldots$ is very dissimilar to that of objective function values.

6 Conclusion and Future Work

We have demonstrated that time-intensive analytics such as clustering can be calibrated to yield information in near real-time due to an empirically observed almost-everywhere local convergence property. This real-time information can enable users to conduct their workflows without waiting for the analytic to converge everywhere.

This work was motivated by real-world applications of clustering in EIM. In particular, we are intrigued by the possible applications of the techniques of this paper to cluster-based retrieval over large document corpora. Abstractly, we have a ranking of clusters based on their convergence. We also have a retrieval ranking of clusters based on their relevance to some information need. Can a meaningful merger of these two rankings be done to provide the user with the most relevant information to their need, as quickly as it is available?

Acknowledgement

Some of the data in this paper was gathered using scripts written by Hernan Laffitte.

References

[1] BOTTOU, L., AND BENGIO, Y. Convergence properties of the k-means algorithm. In *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference* (Cambridge, Massachusetts, 1995), G. Tesauro, D. Touretzky, and T. Leen, Eds., MIT Press, pp. 585–592.

[2] CUTTING, D. R., PEDERSEN, J. O., KARGER, D., AND TUKEY, J. W. Scatter/gather: A cluster-based approach to browsing large document collections. In *Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1992), pp. 318–329.

[3] HEARST, M. A., AND PEDERSEN, J. O. Re-examining the cluster hypothesis: Scatter/gather on retrieval results. In *Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval* (New York, 1996), ACM Press, pp. 76–84.

[4] KARYPIS, G., AND HAN, E.-H. S. Fast supervised dimensionality reduction algorithm with applications to document categorization & retrieval. In *Proceedings of the ninth international conference on Information and knowledge management* (New York, NY, USA, 2000), CIKM '00, ACM, pp. 12–19.

[5] LAGUS, K., HONKELA, T., KASKI, S., AND KOHONEN, T. Self-organizing maps of document collections: A new approach to interactive exploration. In *KDD* (1996), pp. 238–243.

[6] PIROLLI, P., SCHANK, P., HEARST, M., AND DIEHL, C. Scatter/gather browsing communicates the topic structure of a very large text collection. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems* (1996), vol. 1 of *PAPERS: Interactive Information Retrieval*, pp. 213–220.

[7] SALTON, G., AND MCGILL, M. Introduction to modern information retrieval. McGraw-Hill, 1983.

[8] SINGHAL, A., BUCKLEY, C., MITRA, M., AND SALTON, G. Pivoted document length normalization. Technical Report TR95–1560, Department of Computer Science, Cornell University, Nov. 1995.

[9] STREHL, A., GHOSH, J., AND MOONEY, R. J. Impact of similarity measures on web-page clustering. In *Proc. AAAI Workshop on AI for Web Search (AAAI 2000), Austin* (July 2000), AAAI/MIT Press, pp. 58–64.

[10] WU, X., KUMAR, V., ROSS QUINLAN, J., GHOSH, J., YANG, Q., MOTODA, H., MCLACHLAN, G. J., NG, A., LIU, B., YU, P. S., ZHOU, Z.-H., STEINBACH, M., HAND, D. J., AND STEINBERG, D. Top 10 algorithms in data mining. *Knowl. Inf. Syst. 14*, 1 (Dec. 2007), 1–37.

AutoTune: Optimizing Execution Concurrency and Resource Usage in MapReduce Workflows[*]

Zhuoyao Zhang
University of Pennsylvania
zhuoyao@seas.upenn.edu

Ludmila Cherkasova
Hewlett-Packard Labs
lucy.cherkasova@hp.com

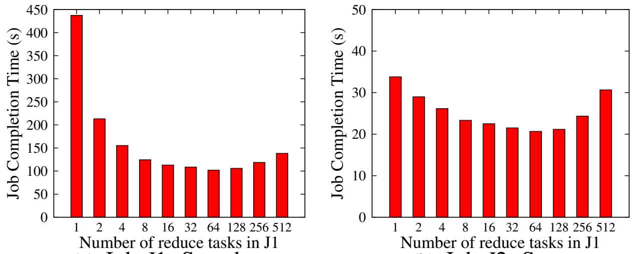
Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Abstract

An increasing number of MapReduce applications are written using high-level SQL-like abstractions on top of MapReduce engines. Such programs are translated into MapReduce workflows where the output of one job becomes the input of the next job in a workflow. A user must specify the number of reduce tasks for each MapReduce job in a workflow. The reduce task setting may have a significant impact on the execution concurrency, processing efficiency, and the completion time of the worklflow. In this work, we outline an automated performance evaluation framework, called *AutoTune*, for guiding the user efforts of tuning the reduce task settings in MapReduce sequential workflows while achieving performance objectives. We evaluate performance benefits of the proposed framework using a set of realistic MapReduce applications: *TPC-H* queries and custom programs mining a collection of enterprise web proxy logs.

1 Introduction

Many companies are embracing MapReduce environments for advanced data analytics over large datasets. Optimizing the execution efficiency of these applications is a challenging problem that requires the user experience and expertize. Pig [4] and Hive [10] frameworks offer high-level SQL-like languages and processing systems on top of MapReduce engines. These frameworks enable complex analytics tasks (expressed as high-level declarative abstractions) to be compiled into *directed acyclic graphs* (DAGs) and *workflows* of MapReduce jobs. Currently, a user must specify the number of reduce tasks for each MapReduce job (the default setting is 1 reduce task). Determining the right number of reduce tasks is non-trivial: it depends on the input sizes of the job, on the Hadoop cluster size, and the amount of resources available for processing this job. In the MapReduce workflow, two sequential jobs are data dependent: the output of one job becomes the input of the next job, and therefore, the number of reduce tasks in the previous job defines the number (and size) of input files of the next job, and may affect its performance and processing efficiency in unexpected ways. To demonstrate these issues we use a *Grep* program provided with the Hadoop distribution. This program consists of a workflow with two sequential jobs: the first job ($J1$) searches for strings with the user-specified patterns and counts them for each matched pattern. The second job ($J2$) reads the output of the first job and sorts the patterns according to their appearance frequencies. We execute this program with 15 GB of input data on a Hadoop cluster with 64 worker nodes. Each node is configured with 2 map and 1 reduce slots, i.e., with 128 map and 64 reduce slots overall. Figure 1 shows the execution times of both jobs $J1$ and $J2$ as we vary the number of reduce tasks in $J1$. (The number of reduce task for $J2$ is fixed to 1).



(a) Job $J1$: Search (b) Job $J2$: Sort

Figure 1: A Motivating Example.

Figure 1 (a) shows that the $J1$ execution time significantly depends on the number of reduce tasks. A low number of reduce tasks limits the job execution concurrency and leads to an increased job completion time. A high number of reduce tasks improves the execution parallelism and the job processing time, but at some point (e.g., 512 reduce tasks), it leads to an increased overhead and increased job processing time.

As the outputs generated by $J1$ become inputs of $J2$, the number of output files and their sizes may have a significant impact on the performance of J_2. Figure 1 (b) shows how the reduce task settings of $J1$ impact the completion time of $J2$.

In this work, we outline a novel performance evaluation framework, called *AutoTune*, that automates the user efforts of tuning the numbers of reduce tasks along the MapReduce workflow. It consist of the following key components:

- *The ensemble of performance models* that orchestrates

[*]This work was mostly done uring Z. Zhang's internship at HP Labs. Prof. B. T. Loo a nd Z. Zhang are supported in part by NSF grants CNS-1117185 and CNS-0845552.

the prediction of the workflow completion time at the cluster and application levels.

- *Optimization strategies* that are used for determining the numbers of reduce tasks along the jobs in the MapReduce workflow for achieving the performance objectives and for analyzing the performance trade-offs.

We validate the accuracy, efficiency, and performance benefits of the proposed framework using a set of realistic MapReduce applications executed on 66-nodes Hadoop cluster. This set includes TPC-H queries and custom programs mining a collection of enterprise web proxy logs. Our case study shows that the proposed ensemble of models accurately predicts workflow completion time. Moreover, the proposed framework enables users to analyze the efficiency trade-offs. Our experiments show that in many cases, by allowing 5%-10% increase in the workflow completion time one can gain 40%-90% of resource usage savings. The ability to optimize the amount of resources used by the programs enables efficient workload management in the cluster.

This paper is organized as follows. Section 2 presents the problem definition and outlines our solution. Sections **??**-3 describe the ensemble of performance models and optimization strategies. Section 4 evaluates the framework accuracy and effectiveness of optimization strategies. Section 5 outlines related work. Section 6 presents conclusion and future work directions.

2 *AutoTune* Solution Outline

Currently, a user must specify the number of reduce tasks for each MapReduce job in a workflow (the default setting is 1 reduce task). The reduce task setting defines the number of paraller tasks that are created for processing data at the reduce stage and the amount of data which is processed and written by each task. As a result, the reduce tasks settings impact the efficiency of the map stage processing in the next job. If too many small output files created it leads to a higher processing overhead and a higher number of map slots is needed for these files processing. Our **main goal** is to determine the reduce tasks settings that *optimize the workflow overall completion time*. Typically, the Hadoop cluster is shared by multiple users and their jobs are scheduled with Fair or Capacity Schedulers. Under these schedulers the cluster resources are partitioned into pools with separate queues and priorities for each pool. The unused cluster resources can be allocated to any pool(s) with jobs that can utilize these resources. Therefore, an **additional goal** is to *minimize the workflow resource usage* for achieving this optimized time. Often nearly optimal completion time can be achieved with a significantly smaller amount of resources (compare the outcome of 32 and 64 reduce task settings in Figure 1).

AutoTune solution relies on the following *Pairwise Optimization Theorem*: the optimization problem of the entire workflow can be efficiently solved through the optimization problem of the pairs of its sequential jobs.

Proof: Figure 2 shows a workflow that consists of three sequential jobs: J_1, J_2, and J_3. To optimize the workflow com-

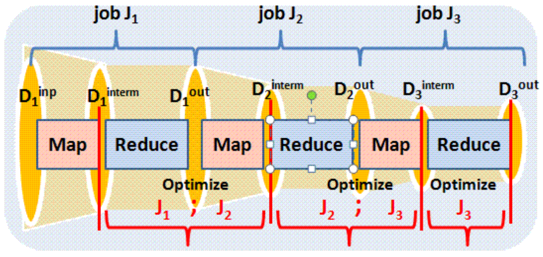


Figure 2: Example workflow with 3 sequential jobs

pletion time we need to tune the reduce task settings in jobs J_1, J_2, and J_3. A *question* to answer is whether the choice of reduce task setting in job J_1 impacts the choice of reduce task setting in job J_2, etc.

A *critical observation* here is that *the size of overall data* generated between the map and reduce stages of the same job and between two sequential jobs *does not depend on the reduce task settings* of these jobs. For example, the overall amount of output data D_1^{out} of job J_1 does not depend on the number of reduce tasks in J_1. It is defined by the size and properties of D_1^{interm}, and the semantics of J_1's reduce function. Similarly, the amount of D_2^{interm} is defined by the size of D_1^{out}, properties of this data, and the semantics of J_2's map function. Again, the size of D_2^{interm} does not depend on the number of reduce tasks in J_1. Therefore the amount of intermediate data generated by the map stage of J_2 is the same (i.e., *invariant*) for different settings of reduce tasks in the previous job J_1. It means that the choice of an appropriate number of reduce tasks in job J_2 does not depend on the choice of reduce task setting of job J_1. It is primarily driven by an optimized execution of the next pair of jobs J_2 and J_3. Finally, tuning the reduce task setting in J_3 is driven by optimizing its own completion time.■

In such a way, the optimization problem of the entire workflow can be efficiently solved through the **optimization problem of the pairs of its sequential jobs**. Therefore, for two sequential jobs J_1 and J_2, we need to design a model that evaluates the execution times of J_1's reduce stage and J_2's map stage as a function of a number of reduce tasks in J_1. Such a model will enable us to iterate through a range of reduce tasks' parameters and identify a parameter that leads to the minimized completion time of these jobs and evaluate the amount of utilized resources (both reduce and map slots used over time).

While there were quite a few research efforts to design a model for predicting the completion time of a MapReduce job [6, 5, 11, 12, 13], it still remains a challenging research problem. The **main challenge** is to estimate the durations of map and reduce tasks (and the entire job) when these tasks process different amount of data (compared to past job runs).

Some earlier modeling efforts for predicting the job completion time analyze map and reduce task durations [12] from the past job runs, and then derive some scaling factors for task execution times when the original MapReduce application is applied for processing a larger dataset [13, 11]. Some other efforts [6, 5, 11] aim to perform a more detailed (and more expensive) job profiling and time prediction at a level of phases that comprise the execution of map and reduce tasks.

In this work, we pursue a new approach for designing a MapReduce performance model that can efficiently predict the completion time of a MapReduce application for processing different given datasets as a function of allocated resources. It combines the useful rationale of the detailed phase profiling method [6] for estimating durations of map/reduce tasks with fast and practical analytical model designed in [12]. However, our new framework proposes a very *different approach* to estimate the execution times of these job phases. We observe that the executions of map and reduce tasks consist of specific, well-defined data processing phases. Only map and reduce functions are custom and their computations are user-defined for different MapReduce jobs. The executions of the remaining phases are *generic*, i.e., strictly regulated and defined by the Hadoop processing framework. The execution time of each generic step depends on the amount of data processed by the phase and the performance of underlying Hadoop cluster. In the earlier papers [6, 5, 11], profiling is done for all the phases (including the generic ones) for each application separately. Then these measurements are used for predicting a job completion time.

In our work, we design *a set of parameterizable microbenchmarks* [16] to measure generic phases and to derive *a platform performance model* of a given Hadoop cluster. We distinguish *five* phases of the map task execution and *three* phases of the reduce task execution as shown in Figure 3.

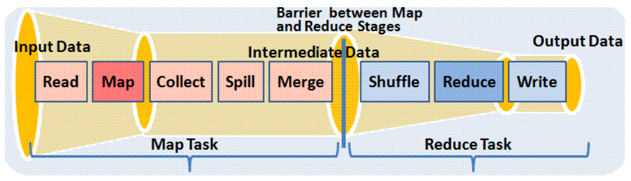


Figure 3: MapReduce Processing Pipeline.

We concentrate on profiling of generic (non-customized) phases of the *MapReduce processing pipeline* (opposite to phase profiling of specific MapReduce jobs). By running a set of diverse benchmarks on a given Hadoop cluster we collect a useful training set (that we call a *platform profile*) that characterizes the execution time of different phases while processing different amounts of data. This profiling can be done in a small test cluster with the same hardware and configuration as the production cluster. Using the created training set and a robust linear regression we derive a *platform performance model* $\mathscr{M}_{platform} = (\mathscr{M}_{read}, \mathscr{M}_{collect}, \mathscr{M}_{spill}, \mathscr{M}_{merge}, \mathscr{M}_{shuffle}, \mathscr{M}_{write})$ that estimates each phase duration as a function of processed data:

$$T^J_{phase} = \mathscr{M}_{phase}(Data^J_{phase}), \tag{1}$$

where $phase \in \{read, collect, spill, merge, shuffle, write\}$.

The proposed evaluation framework aims to **divide** *i)* the performance characterization of the underlying Hadoop cluster and *ii)* the extraction of specific performance properties of different MapReduce applications. It aims to derive **once** an accurate performance model of Hadoop's generic execution phases as a function of processed data, and then **reuse** this model for characterizing performance of generic phases across different applications (with different job profiles).

For profiling map and reduce phases (user-defined map and reduce functions) of production MapReduce jobs we apply our alternative profiling tool based on *BTrace* approach [2]. It can be applied to jobs in the production cluster. Since we only profile map and reduce phase execution – the overhead is small.

Once we approximate the execution times of map and reduce tasks, we could model the completion time of a single job by applying the analytical model designed in ARIA [12]. The proposed performance model utilizes the knowledge about average and maximum of map/reduce task durations for computing the lower and upper bounds on the job completion time as a function of allocated map and reduce slots. Equation 2 shows the lower-bound on the job completion time:

$$T^{low}_J = \frac{N^J_M \cdot M^J_{avg}}{S^J_M} + \frac{N^J_R \cdot R^J_{avg}}{S^J_R} \tag{2}$$

where M^J_{avg} (R^J_{avg}) represent the average map (reduce) task duration, N^J_M (N^J_R) denote the map (reduce) task number and S^J_M (S^J_R) reflect the number of map (reduce) slots for processing the job. The computation of the upper bound on the job completion time is slightly different (see [12] for details: the formula involves the estimates of maximum map/reduce task durations). As it is shown in [12], the average of lower and upper bounds serves as a good prediction of the job completion time (it is within 10% of the measured one).

3 Two Optimization Strategies

According to *Pairwise Optimization Theorem* the optimization problem of reduce task settings for a given workflow $W = \{J_1, ..., J_n\}$ can be efficiently solved via the *optimization problem of the pairs of its sequential jobs*. Therefore, for any two sequential jobs (J_i, J_{i+1}), where $i = 1, ..., n-1$, we need to evaluate the execution times of J_i's reduce stage and J_{i+1}'s map stage as a function of the number of reduce tasks $N^{J_i}_R$ in J_i (see the related illustration in Figure 2, Section 2). Let us denote this execution time as $T_{i,i+1}(N^{J_i}_R)$.

By iterating through the number of reduce tasks in J_i we can find the reduce task setting $N^{J_i,min}_R$ that results in the minimal completion time $T^{min}_{i,i+1}$ for the pair (J_i, J_{i+1}), i.e., $T^{min}_{i,i+1} = T_{i,i+1}(N^{J_i,min}_R)$. By determining the reduce task settings s for all the job pairs, i.e., $s^{min} = \{N^{J_1,min}_R, ..., N^{J_n,min}_R\}$, we can determine the minimal workflow completion time $T_W(s^{min})$. *AutoTune* can be used with Hadoop Fair Scheduler or Capacity Scheduler and multiple jobs executed on a cluster. Both schedulers allow configuring different size resource pools each running jobs in the FIFO manner. Note, that the proposed approach for finding the reduce task setting that minimizes the workflow completion time can be applied to a different amount of available resources, e.g., the entire cluster or a fraction of available cluster resources. Therefore, the optimized workflow execution can be constructed for any size resource pool managed (available) in a Hadoop cluster.

We aim to design the optimization strategy that enables a user to analyze the possible trade-offs, such as workflow performance versus its resource usage. We aim to *answer the following question*: if the performance goal allows a specified increase of the minimal workflow completion time $T_W(s^{min})$, e.g., by 10%, then what is the resource usage under this workflow execution compared to $R_W(s^{min})$?

We define the resource usage $R_{i,i+1}(N_R^{J_i})$ for a sequential job pair (J_i, J_{i+1}) executed with the number of reduce tasks $N_R^{J_i}$ in job J_i as follows:

$$R_{i,i+1}(N_R^{J_i}) = T_{R\_task}^{J_i} \times N_R^{J_i} + T_{M\_task}^{J_{i+1}} \times N_M^{J_{i+1}}$$

where $N_M^{J_{i+1}}$ represent the number of map tasks of job J_{i+1}, and $T_{R\_task}^{J_i}$ and $T_{M\_task}^{J_{i+1}}$ represent the average execution time of reduce and map tasks of J_i and J_{i+1} respectively. The resource usage for the entire MapReduce workflow is defined as the sum of resource usage for each job within the workflow.

Table 1 summarizes the notations that we use for defining the optimization strategies below.

Table 1: Notation Summary

$T_{i,i+1}(N_R^{J_i})$	Completion time of (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$R_{i,i+1}(N_R^{J_i})$	Resource usage of pair (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$T_W(s)$	Completion time of the entire workflow W with setting s
$R_W(s)$	Resource usage of the entire workflow W with setting s
$T_{i,i+1}^{min}$	Minimal completion time of a job pair (J_i, J_{i+1})
$N_R^{J_i,min}$	Number of reduce tasks in J_i that leads to $T_{i,i+1}^{min}$
$w\_increase$	Allowed increase of the min workflow completion time
$N_R^{J_i,incr}$	Number of reduce tasks in J_i to meet the increased time

The first algorithm is based on the *local optimization*. The user specifies the allowed increase *w_increase* of the minimal workflow completion time $T_W(s^{min})$. Our goal is to compute the new workflow reduce task settings that allow achieving this increased completion time. To accomplish this goal, a straightforward approach is to apply the user-defined *w_increase* to the minimal completion time $T_{i,i+1}^{min}$ of each pair of sequential jobs (J_i, J_{i+1}), and then determine the corresponding number of reduce tasks in J_i. The pseudo-code defining this strategy is shown in Algorithm 1. The completion time of each job pair is locally increased (line 2), and then the reduce task settings are computed (lines 4-6).

While this local optimization strategy is simple to implement, there could be additional resource savings achieved if we consider a *global optimization*. Intuitively, the resource usage for job pairs along the workflow might be quite different depending on the job characteristics. Therefore, we could identify the job pairs with the highest resource savings (gains) for their increased completion times. The pseudo-code defining this global optimization strategy is shown in Algorithm 2.

Algorithm 1 Local optimization strategy for deriving workflow reduce tasks' settings

```
1: for $i \leftarrow 1$ to $n$ do
2:     $T_{i,i+1}^{incr} = T_{i,i+1}^{min} \times (1 + w\_increase)$
3:     $N_R^{J_i,cur} \leftarrow N_R^{J_i,min}$
4:     while $T_{i,i+1}(N_{R_i}^{J_i,cur}) < T_{i,i+1}^{incr}$ do
5:         $N_R^{J_i,cur} \leftarrow N_R^{J_i,cur} - 1$
6:     end while
7:     $N_R^{J_i,incr} \leftarrow N_R^{J_i,cur}$
8: end for
```

Algorithm 2 Global optimization strategy for deriving workflow reduce tasks' settings

```
1: $s^{cur} = s^{min} = \{N_R^{J_1,min}, ..., N_R^{J_n,min}\}$
2: $T_{w\_incr} = T_W(s^{min}) \times (1 + w\_increase)$
3: for $i \leftarrow 1$ to $n$ do
4:     $N_R^{J_i,incr} \leftarrow N_R^{J_i,min}$
5: end for
6: while true do
7:     $bestJob = -1, \quad maxGain = 0$
8:     for $i \leftarrow 1$ to $n$ do
9:         $N_R^{J_i,tmp} \leftarrow N_R^{J_i,incr} - 1$
10:        $s^{tmp} = s^{cur} \cup \{N_R^{J_i,tmp}\} - \{N_R^{J_i,incr}\}$
11:        if $T_W(s^{tmp}) \le T_{w\_incr}$ then
12:            $Gain = \frac{R_W(s^{min}) - R_W(s^{tmp})}{T_W(s^{tmp}) - T_W(s^{min})}$
13:            if $Gain > MaxGain$ then
14:                $maxGain \leftarrow Gain, \quad bestJob \leftarrow i$
15:            end if
16:        end if
17:    end for
18:    if $bestJob = -1$ then
19:        break
20:    else
21:        $N_R^{bestJob,incr} \leftarrow N_R^{bestJob,incr} - 1$
22:    end if
23: end while
```

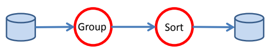
First, we apply the user-specified *w_increase* to determine the targeted completion time T_{w_incr} (line 2). The initial number of reduce task for each job J_i is set to $N_R^{J_i,min}$ (lines 3-5), and then we go through the iteration that at each round estimates the *gain* we can get by decreasing the number of reduce tasks by one for each job J_i. We aim to identify the job that has the smallest response time increase with the decreased amount of reduce tasks while satisfying the targeted workflow completion time (lines 8-17). We pick the job which brings the largest gain and decrease its reduce task setting by 1 (line 21). Then the iteration repeats until the number of reduce tasks in any job cannot be further decreased because it would cause a violation of the targeted workflow completion time T_{w_incr} (line 11).

4 Evaluation

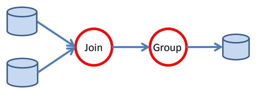
Experimental Testbed and Workloads. All experiments are performed on 66 HP DL145 G3 machines. Each machine has four AMD 2.39GHz cores, 8 GB RAM and two 160 GB 7.2K rpm SATA hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We use Hadoop 1.0.0 and Pig-0.7.0 with two machines dedicated to*JobTracker* and *NameNode*, and remaining 64 machines as workers. Each worker is configured with 2 map and 2 reduce slots. The HDFS blocksize is set to 64MB. The replication level is set to 3. We disabled speculative execution since it did not lead to significant improvements in our experiments.

To validate the accuracy, effectiveness, and performance benefits of the proposed framework, we use queries from TPC-H benchmark and custom queries mining a collection of web proxy logs. TPC-H [3] is a standard database benchmark for decision-support workloads. It comes with a data generator for creating the test database. The input dataset size is controlled by the scaling factor: the scaling factor of 1 generates 1 GB input dataset. Our second dataset contains 6 months access logs of the enterprise web proxy during 2011-2012 years.

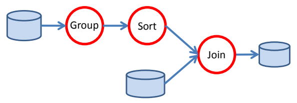
TPC-H and proxy queries are implemented using Pig [4]. Queries are translated into *sequential* MapReduce workflows that are graphically represented in Fig. 4.



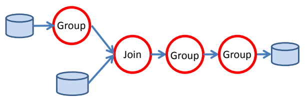
(a) TPC-H Q1



(b) TPC-H Q19



(c) proxy-Q1

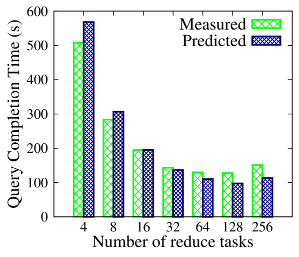


(d) proxy-Q2

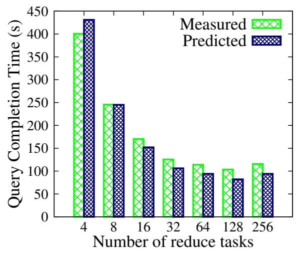
Figure 4: Workflows for TPC-H and Proxy queries.

AutoTune Performance Benefits. Since it is infeasible to validate optimal settings by testbed executions (unless we exhaustively execute the programs with all possible settings), we evaluate the models' accuracy to justify the optimal settings procedure.

We execute two queries *TPC-H Q1* and *TPC-H Q19* with the total input size of 10 GB in our 66-node Hadoop cluster. Figure 5 shows measured and predicted query completion times for a varied number of reduce tasks in the first job of both workflows (the number of reduce tasks for the second job is fixed in these experiments). First of all, results presented in Figure 5 reflect a good quality of our models: the difference between measured and predicted completion times for most of the experiments is less than 10%. Moreover, the predicted completion times accurately reflect a similar trend observed in measured completion times of the studied workflows as a function of the reduce task configuration. These experiments demonstrate that there is a significant difference (up to 4-5 times) in the workflow completion times depending on the reduce task settings.



(a) *TPC-H Q1*



(b) *TPC-H Q19*

Figure 5: Model validation for *TPC-H Q1* and *TPC-H Q19*.

Figure 5 shows that the query completion time decreases with the increased number of reduce tasks (because it leads to a higher concurrency degree and a smaller amount of data processed by each task). However, at some point job settings with a high number of reduce tasks (e.g., 256) may have a negative effect due to higher overheads and higher resource allocation required to process such a job.

Another interesting observation from the results in Figure 5 is that under two settings with a number of reduce tasks equal to 64 and 128 the workflow completion times are very similar while the number of required reduce slots for a job execution increases twice. As shown later in this section, *AutoTune* enables the user to identify useful trade-offs in achieving the optimized workflow completion time while minimizing the amount of resources required for a workflow execution.

Why Not Use Best Practices? There is a list of best practices [1] that offers useful guidelines in determining the appropriate configuration settings. The widely used *rule of thumb* suggests to set the number of reduce tasks to 90% of all available resources (reduce slots) in the cluster. Intuitively, this maximizes the concurrency degree in job executions while leaving some "room" for recovering from the failures. This approach may work under the FIFO scheduler when all the cluster resource are (eventually) available to the next scheduled job. This guideline does not work well when the Hadoop cluster is shared by multiple users, and their jobs are scheduled with Fair or Capacity Schedulers. Moreover, the rule of thumb suggests the same number of reduce tasks for all applications without taking into account the amount of input data for processing in these jobs.

To illustrate these issues, Figure 6 shows the impact of the number of reduce tasks on the measured query completion time for executing *TPC-H Q1* and *TPC-H Q19* with different input dataset sizes. The *rule of thumb* suggests to use 115 reduce tasks (128*0.9=115). However, as we can see from the results in Figure 6 (a), for dataset sizes of 10 GB and 15 GB the same performance could be achieved with 50% of the suggested resources. The resource savings are even higher for *TPC-H Q1* with 5 GB input size: it can achieve the nearly optimal performance by using only 24 reduce tasks (this represents 80% savings against the *rule of thumb* setting). The results for *TPC-H Q19* show similar trends and conclusion.

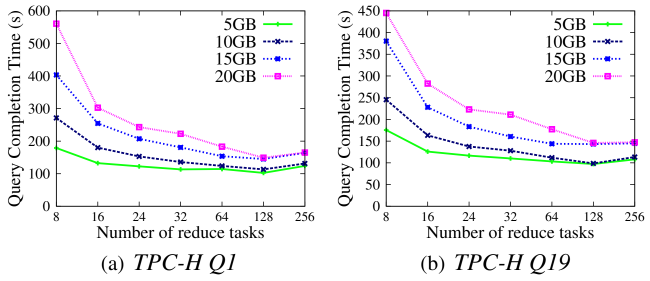


Figure 6: Effect of reduce task settings for processing the same job with different input dataset sizes (measured results).

In addition, Figure 7 shows the effect of reduce task settings on the measured completion time of *TPC-H Q1* query when only a fraction of resources (both map and reduce slots) is available for the job execution. Figures 7 (a) and (b) show

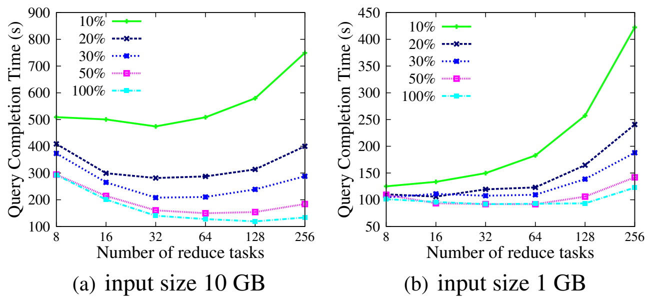


Figure 7: Effect of reduce task settings when only a fraction of resources is available (measured results).

the results with the input dataset size of 10 GB and 1 GB respectively. The graphs reflect that when less resources are available to a job (e.g., 10% of all map and reduce slots in the cluster), the offered rule of thumb setting (115 reduce tasks) could even hurt the query completion time since the expected high concurrency degree in the job execution cannot be achieved with limited resources while the overhead introduced by a higher number of reduce tasks causes a longer completion time. This negative impact is even more pronounced when the input dataset size is small as shown in Figure 7 (b). For example, when the query can only use 10% of cluster resources, the query completion time with the rule of thumb setting (115 reduce tasks) is more than 2 times higher compared to the completion time of this query with eight reduce tasks.

Analyzing Performance Trade-offs. Now, we evaluate two optimization strategies introduced in Section 3 for deriving workflow reduce task settings and analyzing the achievable performance trade-offs. Figure 8 presents the normalized resource usage under local and global optimization strategies when they are applied with different thresholds for a workflow completion time increase, i.e., *w_increase= 0%, 5%, 10%, 15%*. Figures 8 (a)-(b) show the *measured results* for two TPC-H queries with the input size of 10GB (i.e., scaling factor of 10), and Figures 8 (c)-(d) show results for two proxy queries that process 3-month data of web proxy logs. For presentation purposes, we show the normalized workflow resource usage with respect to the resource usage under the *rule of thumb* setting that sets the number of reduce tasks in the job to 90% of the available reduce slots in the cluster. In the presented results, we also eliminate the resource usage of the map stage in the first job of the workflow as its execution does not depend on reduce task settings.

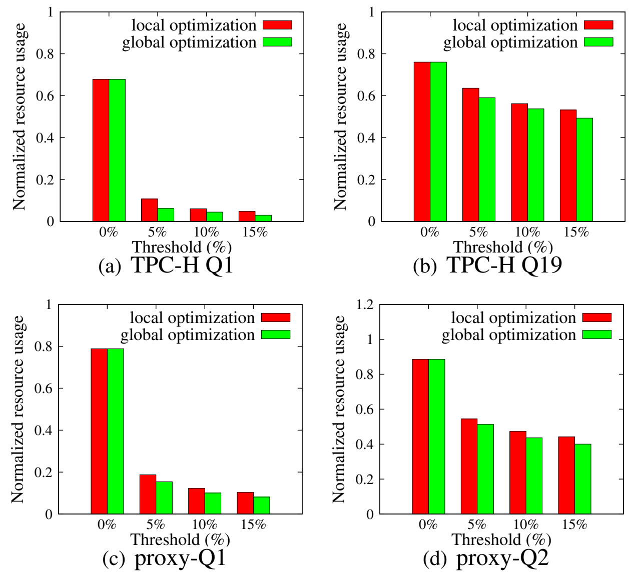


Figure 8: Local and global optimization strategies: resource usage with different *w_increase* thresholds.

The results are quite interesting. The first group of bars in Figure 8 shows the normalized resource usage when a user aims to achieve the minimal workflow completion time (*w_increase= 0%*). Even in this case, there are 5%-30% resource savings compared to the *rule of thumb* settings. When *w_increase= 0%* the local and global optimization strategies are identical and produce the same results. However, if a user accepts 5% of the completion time increase it leads to very significant resource savings: 40%-95% across different queries shown in Figure 8. The biggest resource savings are achieved for *TPC-H Q1* and *Proxy Q1*: 95% and 85% respectively. Moreover, for these two queries global optimization strategy outperforms the local one by 20%-40%. As we can see the performance trade-offs are application dependent.

In summary, *AutoTune* offers a useful framework for a proactive analysis of achievable performance trade-offs to enable an efficient workload management in a Hadoop cluster.

5 Related Work

Several different approaches were proposed for predicting the performance of MapReduce applications [6, 5, 11, 12, 17].

In *Starfish* [6], the authors apply dynamic Java instrumentation to collect a run-time monitoring information about job execution. They create a fine granularity job profile that consists of a diverse variety of metrics. This detailed job profiling enables the authors to predict the job execution under different Hadoop configuration parameters, automatically derive an optimized cluster configuration, and solve cluster sizing problem [5]. Tian and Chen [11] propose predicting a given MapReduce application performance from a set of test runs on small input datasets and a small Hadoop cluster. By executing a variety of 25-60 test runs the authors create a training set for building a model of a given application. It is an interesting approach but the model has to be built for each application and cannot be applied for parameter tuning problems. *ParaTimer* [7] offers a progress estimator for parallel queries expressed as Pig scripts [4]. In the earlier work [8], the authors design *Parallax* – a progress estimator that aims to predict the completion time of a limited class of Pig queries that translate into a sequence of MapReduce jobs. These models are designed for estimating the remaining execution time of workflows and DAGs of MapReduce jobs. However, the proposed models are not applicable for optimization problems.

ARIA [12] builds an automated framework for extracting compact job profiles from the past application run(s). These job profiles form the basis of a *MapReduce analytic performance model* that computes the lower and upper bounds on the job completion time. ARIA provides an SLO-based scheduler for MapReduce jobs with timing requirements. The later work [17] enhances and extends this approach for performance modeling and optimization of Pig programs. In our work, we design a different profiling of MapReduce jobs via eight execution phases. The phases are used for estimating map/reduce tasks durations when the job configuration is modified.

MRShare [9] and *CoScan* [14] offer frameworks that merge the executions of MapReduce jobs with common data inputs in such a way that this data is only scanned once, and the workflow completion time is reduced. AQUA [15] proposes an automatic query analyzer for MapReduce workflow on relational data analysis. It tries to optimize the workflow performance by reconstructing the MapReduce DAGs to minimize the intermediate data generated during the workflow execution.

In our work, we focus on optimizing the workflow performance via tuning the number of reduce tasks of its jobs under a given Hadoop cluster configuration. We are not aware of any published work solving this problem.

6 Conclusion

Optimizing the execution efficiency of MapReduce workflows is a challenging problem that requires the user experience and expertize. In this work, we outline the design of *AutoTune* - the automated framework for tuning the reduce task settings while achieving multiple performance objectives. We observe that the performance gain for minimizing a workflow completion time has a point of diminishing return. In the future, we plan to design useful utility functions for automating the trade-off part of analysis in the optimization process.

References

[1] Apache hadoop: Best practices and anti-patterns. `http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/`, 2010.

[2] BTrace: A Dynamic Instrumentation Tool for Java. `http://kenai.com/projects/btrace`.

[3] TPC Benchmark H (Decision Support), Version 2.8.0, Transaction Processing Performance Council (TPC), http://www.tpc.org/tpch/, 2008.

[4] A. Gates et al. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. of VLDB Endowment*, 2(2), 2009.

[5] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of ACM Symposium on Cloud Computing*, 2011.

[6] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of 5th Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[7] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proc. of SIGMOD*. ACM, 2010.

[8] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of ICDE*, 2010.

[9] T. Nykiel et al. MRShare: sharing across multiple queries in MapReduce. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a Warehousing Solution over a Map-Reduce Framework. *Proc. of VLDB*, 2009.

[11] F. Tian and K. Chen. Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds. In *Proc. of IEEE Conference on Cloud Computing (CLOUD 2011)*.

[12] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. *Proc. of the 8th ACM Intl. Conf. on Autonomic Computing (ICAC)*, 2011.

[13] A. Verma, L. Cherkasova, and R. H. Campbell. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. *Proc. of the 12th ACM/IFIP/USENIX Middleware Conference*, 2011.

[14] X. Wang, C. Olston, A. Sarma, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *Proc. of the ACM Symposium on Cloud Computing,(SOCC'2011)*, 2011.

[15] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[16] Z. Zhang, L. Cherkasova, and B. T. Loo. Benchmarking Approach for Designing a MapReduce Performance Model. In *Proc. of the 4th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*, 2013.

[17] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *Proc. of ICAC*, 2012.

Self-healing and optimizing of the HIP-based M2M overlay network

Amine Dhraief[1], Khalil Drira[2] and Abdelfettah Belghith[1]

[1]HANA Research Group
University of Manouba, Tunisia
{*firist.last*}*@hanalab.org*
[2]LAAS-CNRS, France
Univ. de Toulouse, France
{*first.last*}*@lass.fr*

Abstract

Machine-to-Machine (M2M) paradigm is a novel communication technology under standardization at both the ETSI and the 3GPP. It involves a set of sensors and actuators (M2M devices) communicating with M2M applications via M2M gateways, with no human intervention. For M2M communications trust and privacy are key requirements. This drove us to propose a host identity protocol (HIP) based M2M overlay network, called HBMON, in order to ensure private communications between M2M devices, M2M gateway and M2M applications. In this paper, we first propose to add the self-healing capabilities to the M2M gateways. We enable at the M2M gateway level the REAP protocol, a failure detection and locator pair exploration protocol for IPv6 multihoming nodes. We also add mobility management capabilities to the M2M gateway in order to handle M2M devices mobility. Furthermore, in this paper we add the self-optimization capabilities to the M2M gateways. We also modify the REAP protocol to continuously monitor the overlay paths in order to always select the best available one in term of RTT. We implement our solution on the OMNeT++ network simulator. Results highlight the novel gateway capabilities: it recovers from failures, handle mobility and always select the best available path.

1 Introduction

Embedded systems such as sensors, smart meters and smart cards are experiencing a tremendous proliferation. Several market forecast predict that the number of these devices will soon outnumber the people on earth. According to the Wireless World Research Forum (WWRF), by 2017 we will have 7 trillion wireless devices serving 7 billion people [20]. Juniper Networks predicts that in 2015, the number of connections between embedded equipments will reach over 500 millions [11]. Machine-to-machine (M2M) communication is considered to be an adequate framework to handle the communication between these embedded systems and their corresponding applications. M2M communication is a novel communication technology under standardization at both the European Telecommunications Standardization Institute (ETSI) [10] and the 3rd Generation Partnership Project (3GPP) [19]. M2M communication is based on an autonomous communication between sensors/actuators and correspondent application over the Internet. The M2M architecture introduces a new level of indirection between the sensors/actuators and the application namely the M2M gateway. The M2M gateway aggregates data packets received form sensors and sends them to the M2M application. It generally communicates with M2M devices via short range communication technologies.

Internet is based on the well-known paradigm: ”keep-it simple in the middle, smart at the edge” [18], which survived for the last four decades. Nonetheless, the M2M gateway breaks this paradigm, instead of ”keep-it simple in the middle, smart at the edge”, it shifts the intelligence towards the middle, at the access level. Hence, M2M technologies leads us to imagine and conceive a novel inter-networking architecture. One of the key requirement of M2M communications is the privacy of the collected information. This requirement drove us to build an M2M overlay network over the Internet based on the Host Identity Protocol (HIP) [14, 15], named HBMON (HIP-based M2M Overlay Network) [6]. In this previous work, we have addressed the formation and the maintenance of the overlay.

In this paper, we propose to add the autonomic management of the overlay. We mainly focus on the self-healing and self-optimization autonomic properties. We enable at the M2M gateway level the REAP protocol, a failure detection and locator pair exploration protocol for IPv6 multihoming nodes [1]. Thus, in our overlay, M2M gateways are able to autonomically detect failures of the overlay links and recover from them. We also add to the M2M gateway the mobility management capabilities. M2M devices mobility can be considered as a failure of the first hop and a failure detection and recovery protocol may handle M2M device mobility. Nonetheless, we need to use a specific mobility support to efficiently handle M2M device mobility as we demonstrated in [7]. In our design, M2M gateways are also able to monitor the available overlay paths and dynamically select the best path in term of Round Trip Time (RTT). We implement our solution on the OMNeT++ network simulator. Results show that our solution is able to detect overlay link failures and recover from them. It is also able to self-optimize the selection of the overlay paths.

The remaining of this paper is organized as follows. Section 2 gives an overview of our previous work HBMON [6], then it details the REAP protocol, finally it focuses on the mobility support in the HIP protocol. Section 3 highlights our contribution; namely the self-healing and optimizing of the HIP-based M2M overlay network. Section 4 presents our simulation results. Section 5 concludes the paper.

2 Related works

In this section, we first give an overview of our previous work on M2M overlay network namely the HIP-based M2M overlay network. Then we detail REAP, a failure detection and locator pair exploration protocol for IPv6 multihoming nodes [1]. Finally, we focus on the mobility support in the HIP protocol.

2.1 The HIP-based M2M overlay network

An M2M communication involves an M2M device communicating with an M2M application via M2M gateways, with no human intervention. The first ”*Machine*” in a Machine-to-Machine communication is a device embedding a sensor and an actuator. The second ”*Machine*” is a device which processes the collected information from the sensor and according to these information may remotely control the actuator. The ”*to*” refers to the M2M end-to-end communication network connecting the two machines. M2M devices upload their traffic to an M2M Gateway which aggregates data collected from several M2M devices and sends them to a corresponding M2M gateway or to an M2M application. The M2M application has a middleware layer where data collected from different M2M devices can be presented to the different applications and services to be further processed. M2M application portfolio covers a broad spectrum, ranging from industrial applications, to smart cities, and vehicular technologies. However, in all these applications, M2M communication trust and privacy are key requirements [2].

In order to build a secure M2M network, we proposed in a previous work a HIP-based M2M overlay network called HBMON [6]. Overlay networks are private logical networks built on the top of an existing network infrastructure (Internet for e.g.,). The overlay paradigm breaks the end-to-end principal. Instead of ”*keep-it simple in the middle, intelligent at the edge*” [18], overlay networks move intelligent toward the middle. Overlay networks rely on middle-boxes (such as overlay router) connected through logical links referred as overlay links. Middle-boxes translates on-demand overlay links into Internet paths. Overlay networks are specialized networks such as peer-to-peer networks, Content-delivery networks (CDN), resilient routing networks and enhanced end-to-end security networks [3].

To define and manage our private M2M overlay network we use the Host Identity Protocol (HIP) [14, 15]. HIP introduces a new sub-layer between the transport and the IP layer. The HIP layer decouples end-host identification from its localization. End-hosts are identified with a cryptographic namespace named Host Identity Tag (HIT) while IP addresses are used as end-host locators. HIP introduces a proxy element in the network architecture, the rendezvous server which holds a secure binding between end-hosts IP addresses and their HITs. Finally, HIP is able to manage both mobility and multihoming transparently to upper layer protocols and thus provides session survivability upon end-host mobility or failures in the currently used path [16]. M2M devices within our overlay network may embed several network interfaces associated with distinct access technologies, each one associated with a distinct Internet Service Provider (ISP). Therefore, such M2M devices may be considered as multihomed M2M devices. Furthermore, M2M devices may be embedded in a vehicle to be traced or tracked and so they can be considered as mobile M2M devices, as they change their point of attachment to the network while they move.

In our previous work [6], we focused on the organization and the membership management of the M2M device within the overlay. We also proposed a novel IPv6 address assigning method in order to configure the overlay members with private IPv6 addresses. From an autonomic networking perspectives, we enabled the self-configuration and self-protection properties. The self-configuration properties allows the autonomic system to dynamically adapt itself to the deployment of new components or changes in its environment. In the HBMON, this functionality is provided by the registration functionality of the HIP protocol which allows M2M devices to autonomically register themselves with a rendezvous server and distribute overlay information between overlay members. The self-protection properties main goal is to give the system the possibility to protect itself from intrusion and any hostile behavior. The cryptographic namespace HIT with the private addresses used within the overlay are the features used by the M2M devices in the HBMON to protect themselves from attacks.

2.2 The reachability protocol: REAP

Multihomed terminals have at least two IP addresses configured concurrently, each one associated with a distinct Internet Service Provider (ISP). These terminals are then reachable via different paths [4]. A multihomed terminal can spread its outgoing traffic among the available paths by applying a load sharing or balancing scheduling technique. However, such a scheduling technique has a negative impact on TCP. In fact, TCP segments sent on paths with lower delays may results in out-of-order TCP segments. Upon receiving an out-of-order segment, destination's TCP immediately sends a duplicated acknowledgment. Three duplicated acknowledgments results into the reduction of the TCP congestion window. Therefore, TCP erroneously concludes that duplicated acknowledgments are due to packet losses and enters in a congestion avoidance phase. Hence, multihomed terminal generally consider one path as primary and the alternate paths as backups. If a failure occurs in the primary path, multihomed terminals rehome their ongoing session to a backup path [8, 9]. The IETF has standardized a protocol for failure detection and locator pair exploration protocol for IPv6 multihoming terminals named the reachability protocol (REAP) [1]. The IETF has designed this protocol for the specific use of the Shim6 protocol. Shim6 is a host-centric multihoming management protocol [17].

REAP relies during its functioning on two timers (send timer, keepalive timer) and a state machine assuming that the communicating nodes have a prior knowledge about their locators. REAP starts the send timer whenever a node sends a packets. If this node has not received any packet until the send timer expires, it performs a full reachability exploration. Otherwise, it stops the send timer and starts the keepalive timer. If the node has not sent any packet until the keepalive timer expiry, then it sends a REAP keepalive message to its corresponding peers. If the corresponding peers receives a keepalive message, then it should stop the send timer and starts the keepalive one. The REAP specification recommends that the keepalive timer should be equal to the send timer divided by three. These two timers are mutually exclusive. In other word, the node is either expecting to receive a payload or preparing to send data. So the send timer is stopped when a payload or keepalive message is received and the keepalive timer is stopped when a payload is generated.

When REAP detects a failure, it starts a full reachability exploration in order to find a new bidirectional working address pair using Probe messages to perform the exploration and associates a state to each probe indicating the status of the communication. REAP defines three states. The first state is OPERATIONAL, it indicates that both of peers consider that their communication does not suffer from any failure. The second state is INBOUNDOK, it reflect the case where the peer considers that its communication has apparently no problem, but its correspondent peer has discovered a failure. The third state is EXPLORING, indicates that the peer has just discovered a problem and has not received any packet form its peers while it should has received.

REAP failure recovery procedure is as follows. First, REAP creates a list of all possible pair of addresses by combining the local locator list and the peer locator list and sorts this list according to some priority specified by the user. Then, it switches its state to Exploring and sends four probes successively, a probe every 0.5s. If it does not receive any probe, it retransmits a probe, but this time the retransmission is controlled by a back-off timer. A node in the OPERATIONAL state and receiving a probe having EXPLORING state means that its correspondent peer has not received its outgoing traffic. This peer then sends a probe having an INBOUNDOK state. A peer in the EXPLORING state and receiving an INBOUNDOK probe conclude that its correspondent peer has received its probe and also that the probed locator pair address is bidirectionally reachable. Thus, it sends a probe having an OPERATIONAL state and finally the communication can be resumed.

2.3 HIP mobility support

Before exchanging any data packets, HIP-enabled communicating nodes have to establish a context. The HIP context is established after a four-way handshake control messages I1,R1,I2,R2. This mechanism is based on a secure exchange of cryptographic keys to authenticate communicating hosts [15]. HIP also introduces a registrar element in its architecture: a rendezvous node (RVS) which binds nodes identification with their locations [13]. HIP nodes update their binding in the RVS node upon each change in their network connectivity. The HIP node may also interact with the the RVS element while establishing the HIP context. In fact, when a HIP node wants to establish a HIP association with a node known only by its HIT, it sends the I1 packet to the RVS indicating the Responder HIT. The RVS resolves the destination HIT into an IP address and relays the packet to the destination. After receiving an incoming I1 packet from a RVS, the Responder directly answers the Initiator and the HIP context establishment is then performed [13].

The HIP communication between two hosts is based on a security association (SA) which is established upon the HIP Base Exchange mechanism [15]. A SA is a set of security parameters agreed by two hosts in order to encrypt and authenticate transferred data. However, several SAs may be established between two hosts such as each SA has its own identifier which is called Security Parameter Index (SPI). The main role of HIP layer is to demultiplex incoming packets to host identity tag (HIT) using the SPI value in the packet and to multiplex outgoing packets to the address source and interface according the SPI value in packet. Consequently, in a HIP network, the locator is not only a IP address but also a key indexing the correspondent security association [16]. Thus, when one of two HIP nodes having an ongoing communication changes its current location to another attachment point, it acquires a new IP address and changes the SPI into SA. So, the moving HIP node has to report to the correspondent node about its new locator in order to maintain the HIP SA. In the following, we illustrate how the Host Identity Protocol supports mobility. The basic HIP mobility scenario is illustrated as follows. For setting up the HIP mobility mechanism, there are two ways to be considered; either, mobility with a single SA pair (only one IP address bound to an interface) without re-keying or mobility with a single SA pair with re-keying. In the former case, which is the simplest one, when the mobile host moves and obtains a new IP address, it notifies the correspondent host sending an UPDATE message containing the new IP address in the LOCATOR parameter and the Old SPI and New SPI values in ESP-INFO parameter. When the correspondent host receives the UPDATE packet, it checks the new address and makes it UNVERIFIED in the interim, while the old address is DEPRECATED. Then it acknowledges the mobile host by the second UPDATE message which contains an ECHO REQUEST to validate the new peer address. As well, it includes ESP INFO with Old and New SPIs set to the current outgoing SPI. Lastly, once receiving the second UPDATE message, the mobile node sends the last UPDATE message including an ECHO RESPONSE in order to definitely validate the new address. Indeed, when the correspondent host receives this ECHO RESPONSE, it automatically marks the new address as ACTIVE and removes the old address. For the second case, a new ESP session key will be regenerated. The mobile host sends the UPDATE message containing a new SPI for the incoming SA. The correspondent host upon receiving the UPDATE message, executes the rekey and replies with the a second message containing its own new SPI, then the readdressing proces ends as without re-keying case.

3 Autonomic management of the HBMON

M2M devices, as defined by the ETSI, are sensors or meters that collect data from the environment and upload them to an M2M application [10]. M2M devices and/or M2M gateways are usually equipped with several access technologies associated with distinct ISPs. They are therefore multihomed entities and consequently several overlay paths exists between M2M devices and M2M applications. M2M devices are generally connected to the M2M gateway with short range technologies (ZigBee for e.g.,); whereas, M2M gateways are usually multihomed middle-boxes, equipped with several access technologies. One of the most fundamental constraint that should be satisfied by M2M technology is communication reliability, especially for fault-tolerant oriented applications such as e-Health monitoring. To ensure communication reliability, we add to our architecture failure detection and recovery capabilities along with path monitoring functionalities.

Moreover, according to the targeted application, M2M devices can either be static or mobile nodes. Mobile nodes usually execute a layer 2 (L2) handover which may be followed by a layer 3 (L3) handover. As a result of the L3 handover, current end-host IP addresses is changed to a new topologically correct one. IP addresses have a dual role, they are considered at the same time end-host locator and session identification. Hence, without an adequate support, running transport session are broken as a consequence of a L3 handover. To ensure transport session survivability upon movement, session identification should remain stable while end-hot locator is changed. HIP addresses this issue by introducing a new stable cryptographic Host Identity Tag (HIT) as node identifier [14]. Mobility can be considered as a failure in the first hop of the path between the M2M device and the M2M application. Thus, we can easily manage the M2M device mobility through the REAP protocol, a failure detection and recovery protocol. Nonetheless, we have shown in [7] that managing the mobility with a failure detection and recovery protocol leads to a huge L3 handover latency. Therefore, we rely on the HIP protocol to handle the mobility of our M2M devices within our overlay. From an autonomic networking perspective, the self-healing property includes failures detection and recovery capability as well as mobility management.

HIP already ensures the self-configuring and the self-protecting properties of the autonomic management of our M2M overlay network. In order to provide the remaining properties (self-healing and self-optimization), we propose to add the REAP support in the M2M gateways of our HBMON.

3.1 Self-healing of the HBMON

3.1.1 Failure detection and recovery

In our M2M overlay network, several overlay paths might exist between the gateway and the corresponding M2M applications. This path diversity is highly recommended for specific fault-tolerant system such as security-oriented applications. In order to design a resilient M2M overlay network, we use the REAP protocol to: (i) monitor the existing paths, (ii) detect failures and recover to a new working path. We enable REAP at the gateway level for several reasons. First of all, in our design [6], the overlay architecture is maintained at the gateway, which is viewed form a HIP perspective as Rendezvous node. Second, the overlay link diversity starts at the gateway level as the sensors are usually single-homed entities. Thus, we couple HIP with REAP at the gateway level. We define new parameters in the HIP messages to support the REAP protocol namely "PROBE" and "KEEP ALIVE". They are of type "NOTIFY". The former is exchanged between peers when a failure is detected and the latter is used to monitor unidirectional communications. We add to the HIP two REAP timers, namely send and keepalive timers. If a peer's send timer expires without receiving any incoming packets, the peer assumes that a failure has affected its currently used overlay path and starts exploring the remaining available overlay paths. In unidirectional communications, the peer has to periodically inform its corresponding node that the currently used overlay link is working. When the keepalive timer expires, the peers sends a keepalive message. At the beginning of a communication, the M2M gateway exchanges with the M2M application data packets and eventually keepalive messages. REAP only monitors the currently used overlay link. If REAP detects a failure through the expiry of the send timer, REAP starts the overlay paths explorations. During this exploration, REAP sends probe messages on each available overlay link having the status exploring. The corresponding peer receiving the probe message replies with a probe message indicating the status of the probed overlay link. Upon receiving a probe message with the status inbound OK, REAP replies with a probe with an operational state and switch the ongoing communication to this newly operational overlay link.

3.1.2 Mobility management

To efficiently manage M2M devices mobility, we propose to enhance the HIP rendez-vous server functionalities, embed at the M2M gateway level, to ensure session survivability between HBMON members.

First of all, an M2M device, member of the HBMON, performs a layer 2 (L2) handover. Once the layer 2 connectivity is established, the M2M device receives an IPv6 router advertisement from the new access router and configures a new global IPv6 address. At this stage, both M2M devices corresponding peers and the HBMON rendez-vous servers are not aware about the M2M device new location. To correctly handle the HBMON mobile nodes (HMN) mobility, we introduce in the HIP protocol the following signaling messages. (i) RVS_Discovery: This signaling message allows to discover the nearest HBMON Rendez-vous server (NHRVS). This message is sent in anycast. (ii) HMN_Loc_Up: Contains two main fields; (1) NEW_IP: to report the new HMN's IP address to the correspondent node, (2) CONTEXT_Req: to request the HBMON Context. (iii) Context Update: Once a HMN obtains a new IP address upon moving, it should inform all the rendez-vous server (HRVS) members of the HBMON multicast group about this new IP address. This message is sent in multicast via the old HRVS.

Fig. 1 illustrates this mechanism through an example where a HBMON overlay is established between a HBMON mobile node (HMN) and a HBMON correspondent node (HCN), HMN and HCN have an ongoing communication and the HMN moves to another autonomous systems (AS). This strategy is an enhanced version of the HIP mobility management presented in [16]. Fig. 2 presents the sequence diagram of the exchanged signaling messages for this strategy.

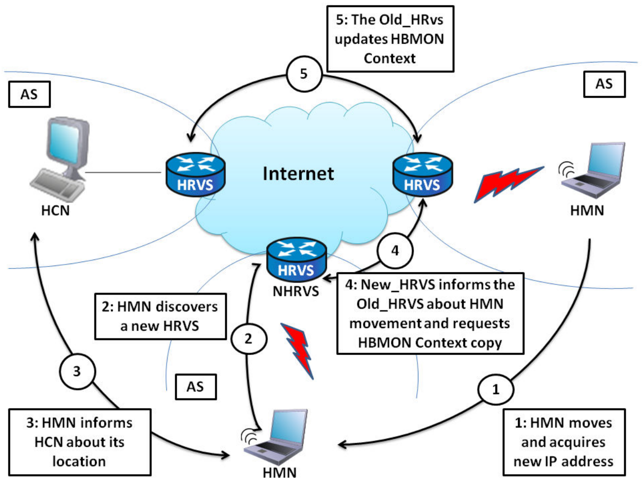

5: The Old_HRvs updates HBMON Context
5
AS
AS
Internet
HRVS
HRVS
HCN
HMN
4
HRVS
NHRVS
4: New_HRVS informs the Old_HRVS about HMN movement and requests HBMON Context copy
2: HMN discovers a new HRVS
2
3
AS
1
3: HMN informs HCN about its location
HMN
1: HMN moves and acquires new IP address

Figure 1: Mobility management scenario

When the HMN moves and acquires a new topologically correct IP address (step 1 Fig. 1), it sends an RVS_Discovery message containing the old HBMON Rendez-vous server's (HRVS's) IP address and its HIT (step 2 Fig. 1). The RVS_Discovery message is sent to a specific anycast address in order to discover the nearest HBMON rendez-vous server (NHRVS). After that, the HMN reports its new IP address to its HCNs using the HIP mobility mechanism; as explained in section 2.3 (step 4 Fig. 1). The new RVS notifies the old HRVS about the new HMN location and triggers the HBMON context update by sending the Context_Req message (step 3 Fig. 1). Once the old HRVS receives a Context_Req message, it updates the mapping between the HMN's HIT and its new IP address. Afterwards, it updates the HBMON context forwarding to all HBMON RVS the Context Update message (step 5 Fig. 1). This message is sent on specific multicast address including all HBMON rendez-vous servers.

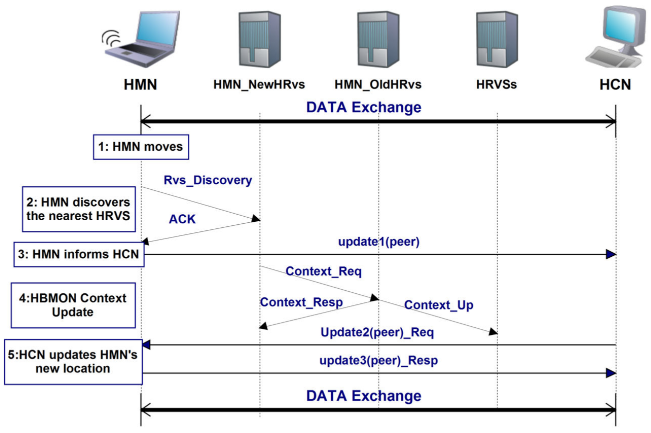


Figure 2: Mobility management sequence diagram

Consequently, we build a self-healing HIP-based M2M overlay network by adding both the failure detection and recovery capability to the M2M gateway, and the M2M device mobility management.

3.2 Self-optimization of the HBMON

The available overlay paths have different network characteristics (RTT, jitter, errors,...) as they cross different Internet Service Providers. An overlay link can experience for a certain period of time a degradation of its network characteristics. Such overlay link can be used by an M2M communication requiring a certain level of quality of service. We propose to use the REAP exploring mechanism to offer to the M2M running communication always the best available link. Instead of triggering the REAP exploring process at the expiry of the send Timer, we continuously monitor the available paths and infer their respective RTT. If the currently used overlay link experience a degradation of its RTT, REAP proposes to HIP a new destination/source address pair of an overlay link having lower RTT. If we frequently perform the inferring of the RTT and overlay paths switching, we can cause overlay paths oscillation, known as route flapping. To avoid route flapping, we add a new timer, namely probe timer which defines the time between two consecutive path exploration. Thus, our HIP-based M2M overlay network is self-optimized as it always selects the best available overlay path in term of RTT.

4 Evaluation

To evaluate our proposal, we use the OMNeT++ simulator coupled with the HIPSim++ framework. We implement the autonomic management of the HBMON in the HIPSim++ framework.

4.1 Failure detection and recovery time

The targeted testbed consists of an M2M device connected to a mutlihomed M2M gateway. The M2M gateway has four available overlay paths having the following RTTs: 50ms, 100ms, 150ms and 200ms. The correspondent node is an M2M application. We set all the wireless accesses to 802.11b at 11Mbit/s. Between the M2M application and the M2M device we use two types of traffic: the first one is an UDP flow having the following characteristics: 20 Bytes the packet length and 40 ms the inter-packet interval, the second traffic is TCP flows, namely an FTP application with hight data traffic. We focus on two metrics: the application recovery time and the instant throughput. The application recovery time (ART) is defined as latency between the last packet received/sent before the outage and the first packet received/sent after the outage.

We evaluate in this section the failure detection and recovery capabilities of our solution. A failure occurs after 20s from the beginning of the communication and lasts twice as the send timer. We measure the ART of UDP traffic and the TCP/FTP traffic. Results are presented by Fig. 3, the x-axis is the send timer value while the y-axis is the measured ART. La Oliva et al [12]. have already measured the ART of both TCP and UDP traffic. By this figures, we aim to validate our REAP implementation in the HIPSim++ framework. We obtain the same results as the one obtained by La Oliva et al. in [12]. Results show that for an UDP application, the ART time increases linearly while we increase the send timer value; whereas, for TCP application the ART experiences several plateaus. After failure recovery, UDP application immediately sends data packets to the newly selected path. Even if a new overlay path is selected, TCP does not send immediately its data segments. TCP has to wait until the TCP Retransmission Timeout (RTO) timer expiry. TCP does not distinguish between a failure recovery process and the congestion in the currently used path [5]. It adjusts the RTO timer as if it has experience of a congestion phase which explains the plateaus in Fig. 3

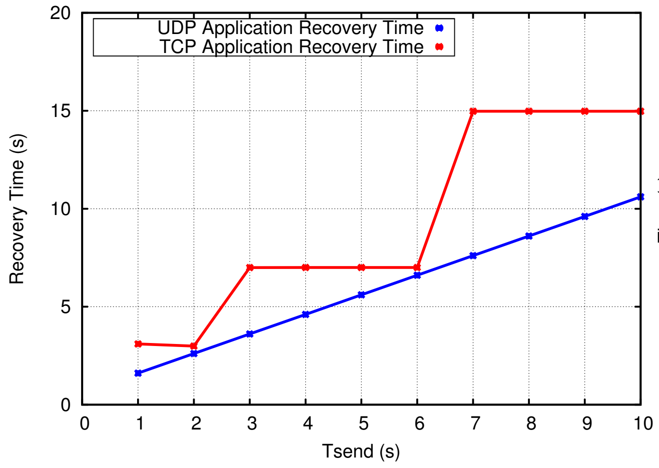


Figure 3: Application recovery time after an outage

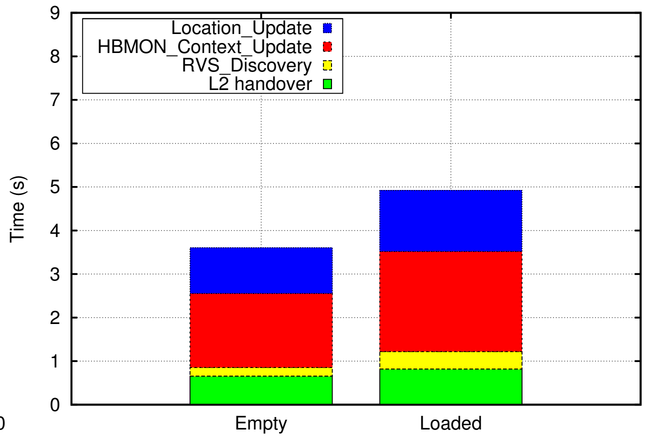


Figure 4: Application recovery time after after M2M device mobility

4.2 Mobility management

We evaluate in this section the mobility management capabilities of our solution. In this scenario, we configure two M2M devices: HMN1 and HMN2, registered respectively with HRVS1 and HRVS2. HMN1 has a 802.11b interface associated with access point AP1 and HMN2 has a 802.11b interface registered with access point AP2. HMN1 is a static node; whereas, HMN2 is a moving according to the random waypoint model. HMN1 and HMN2 exchange a 1 Mbit/s UDP traffic. We load the visited network with three nodes, each of them generating a UDP traffic at 1 Mbit/s In our simulation, we measured the ART which is the latency elapsed between the last packet sent with the old IP address and the first packet sent with the new IP address. The histogram presented in Fig. 4 illustrates the measured ART for an empty and loaded visited network.

The ART latency is decomposed into 4 phases: (i) L2 handover, (ii) RVS_Discovery, (iii) HBMON_Context_Update and (iv) Location_Update. In an empty visited network, the L2 handover latency is 0.65s, the RVS_Discovery latency is 0.2s. The HBMON_Context_Update latency is 1.7s and the Location_Update latency is 1.05s. In a loaded visited network, the L2 handover latency is 0.819s, the RVS_Discovery latency is 0.4s, the HBMON_Context_Update latency is 2.3s and the Location_Update latency is 1.4s. We observe that with our solution, running session effectively resume after the mobility. The mobility singling lasts more than 3.6s for the case of an empty visited network (the best measured case) which is inadequate for real time applications. Nonetheless, M2M applications are usually low data-rate application, and providing session survivability - even after 3.6s of interruption- is preferable than completely losing the currently ongoing session.

4.3 Path exploration

We evaluate in this section the self-optimization capability of our solution. We modify REAP to actively monitor the available paths in order to offer the ongoing M2M communication the best available overlay path in term of RTT.

We focus on the following scenario: the currently used overlay path has an RTT of 50ms and a transient failure affects this path after 20s of the beginning of the M2M communication, the failure lasts the double of the probe timer. Fig. 5 shows the obtained results for a TCP session and a probe timer set to 3s. The x-axis is the time in second and the y-axis is the instant throughput. The obtained results show that during the first 20s, the throughput reaches its maximum because the used path has the minimum RTT (50ms). After the failure recovery, REAP detects a new working overlay path having the second best RTT (100ms). As soon as the best overlay path (50ms) recovers forms its failure, M2M communication switches to this new path and the throughput reaches again its maximum value. Fig. 6 shows the obtained results for a running UDP session and a probe timer set to 3s. The obtained results show the same behavior as for the TCP case in Fig. 5. After the outage, the UDP session is rehomed to a new working ovelray path (100ms). As soon as the new overlay path (50ms) becom ready, the UDP session is rehomed to this newly available path, and the throughput reaches again its maximum value.

In a second scenario, we explore the self-optimization capability of our solution by modifying the load of the currently used overlay path. The M2M communication starts in the overlay path having the lowest RTT. A congestion appears in this path, so the TCP ongoing connection experiences packet losses, TCP reduces its congestion window which impact the instant throughput of the M2M communication. Our solution detects the quality degradation of the path and switches the communication to the second best path in term of RTT. Results presented by Fig. 7 and Fig. 8 shows this dynamic selection of the most stable path. During the first 20s, the M2M communication flows via the path having the lowest RTT (50ms). We inject in this path aggressive UDP traffic, creating therefore a congestion path. Our solution detects the degradation of the RTT of this path and its fluctuations. It switches the ongoing communication to the second path. We repeat the same scenario on this second path. Our solution switches one more time the communication to a third path and finally to the last one until it finds a stable path in term of RTT and packet loss.

From Fig. 5, Fig. 6, Fig. 7 and Fig. 8 we clearly see that we build a self-optimized solution. It is able to detect failure in the currently used overlay path, select a new working path and monitor the remaining paths.

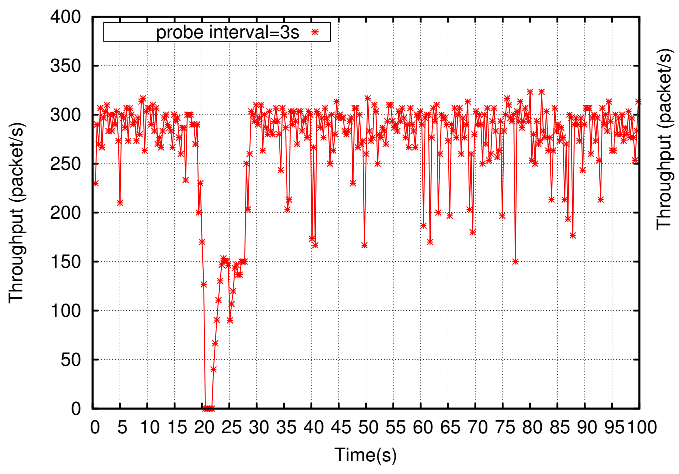


Figure 5: FTP recovery time

5 Conclusion

M2M communication is a new paradigm under standardization at both the ETSI and the 3GPP. This novel technology breaks the end-to-end principle as it introduces a novel element in the network architecture namely the M2M gateway. The M2M gateway aggregates the data collected from the M2M devices and sends them to a correspondent M2M application. In a previous work [6], we have designed a HIP-based M2M overlay network over the existing Internet. This overlay ensures a private communication between M2M devices and their corresponding M2M applications. In this work, we added the autonomic management of our M2M overlay net-

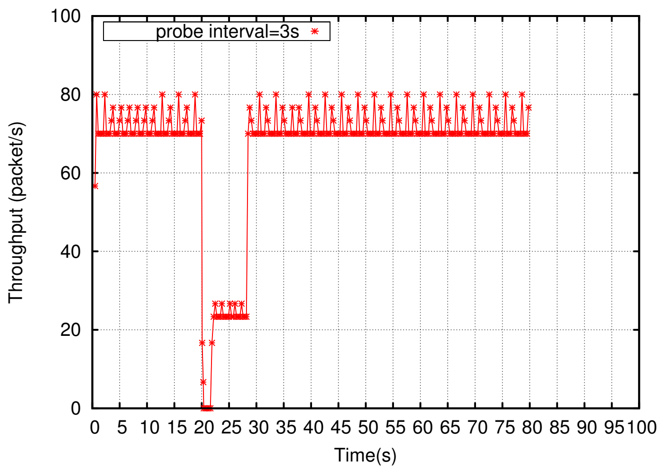


Figure 6: UDP recovery time

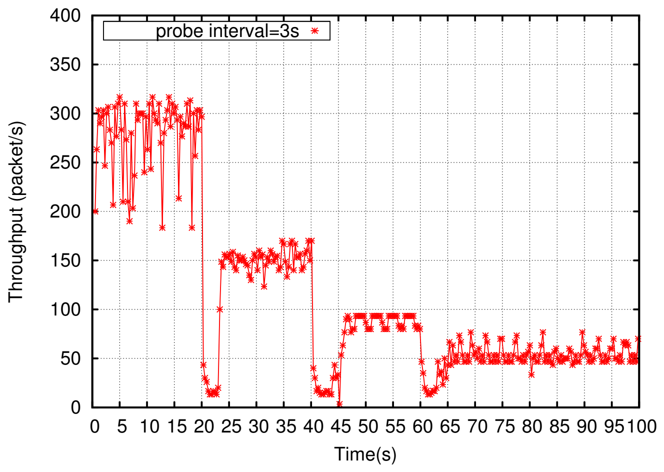


Figure 7: FTP dynamic path selection

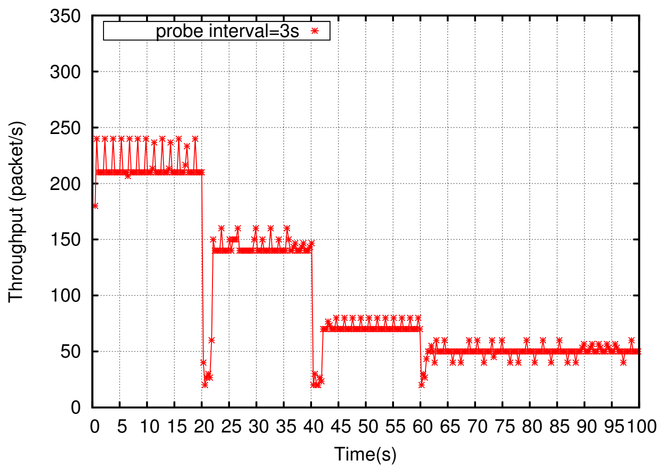


Figure 8: UDP dynamic path selection

work. We focused mainly on the self-healing and the self-optimized autonomic properties. We enhanced the M2M gateway with the failure detection and recovery mechanism, M2M device mobility management and with autonomic path selection capabilities. We implemented and evaluated our solution on the OMNeT++ network simulator. Results shows that the gateway is able to switch from one overlay path to another either due to failure or due to the path characteristic degradation. Results also show that M2M devices running sessions survive to the mobility episode. We are currently implementing this solution on the phidget[1] testbed.

Acknowledgment

This work has been partially funded by the project ITEA2-A2NETS.

References

[1] ARKKO, J., AND VAN BEIJNUM, I. Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. RFC 5534 (Proposed Standard), June 2009.

[2] BAILEY, D. A. Moving 2 Mishap: M2M's Impact on Privacy and Safety. *IEEE Security and Privacy 10*, 1 (2012), 84–87.

[3] CLARK, D., LEHR, B., BAUER, S., FARATIN, P., SAMI, R., AND WROCLAWSKI, J. Overlay Networks and the Future of the Internet. *COMMUNICATIONS & STRATEGIES 63*, 3 (2006).

[4] DHRAIEF, A., AND BELGHITH, A. Multihoming support in the internet: A state of the art. In *International Conference on Models of Information and Communication Systems (MICS 2010)* (2010).

[5] DHRAIEF, A., AND BELGHITH, A. An experimental investigation of the impact of mobile ipv6 handover on transport protocols. *The Smart Computing Revue, KAIS 2*, 1 (2012).

[6] DHRAIEF, A., GHORBALI, M. A., BOUALI, T., BELGHITH, A., AND DRIRA, K. HBMON: A HIP-Based M2M Overlay Network. In *The 2012 Third International Conference on the Network of the Future (NoF 2012)* (2012).

[7] DHRAIEF, A., MABROUKI, I., AND BELGHITH, A. A service-oriented framework for mobility and multihoming support. In *Electrotechnical Conference (MELECON), 2012 16th IEEE Mediterranean* (march 2012), pp. 489 –493.

[8] DHRAIEF, A., AND MONTAVONT, N. Rehoming decision algorithm: Design and empirical evaluation. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 02* (Washington, DC, USA, 2009), CSE '09, IEEE Computer Society, pp. 464–469.

[9] DHRAIEF, A., ROPITAULT, T., AND MONTAVONT, N. Mobility and multihoming management and strategies. In *14th Eunice Open European Summer School 2008* (2008), IFIP, Ed., RSM Dept (Institut TELECOM ;TELECOM Bretagne).

[10] ETSI TECHNICAL COMMITTEE MACHINE-TO-MACHINE COMMUNICATIONS (M2M). Machine-to-Machine communications (M2M); Functional architecture. TS 102 690, Oct. 2011.

[11] JUNIPER NETWORKS. Machine-to-Machine (M2M) The Rise of the Machines. white paper, 2011.

[12] LA OLIVA, A., BAGNULO, M., GARCÍA-MARTÍNEZ, A., AND SOTO, I. Performance analysis of the reachability protocol for ipv6 multihoming. In *Proceedings of the 7th international conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking* (Berlin, Heidelberg, 2007), NEW2AN '07, Springer-Verlag, pp. 443–454.

[13] LAGANIER, J., AND EGGERT, L. Host Identity Protocol (HIP) Rendezvous Extension. RFC 5204 (Experimental), Apr. 2008.

[14] MOSKOWITZ, R., AND NIKANDER, P. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[15] MOSKOWITZ, R., NIKANDER, P., JOKELA, P., AND HENDERSON, T. Host Identity Protocol. RFC 5201 (Experimental), Apr. 2008. Updated by RFC 6253.

[16] NIKANDER, P., HENDERSON, T., VOGT, C., AND ARKKO, J. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206 (Experimental), Apr. 2008.

[17] NORDMARK, E., AND BAGNULO, M. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[18] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 277–288.

[1] http://www.phidgets.com/

[19] TECHNICAL SPECIFICATION GROUP SERVICES AND SYSTEM ASPECTS. Study on facilitating machine to machine communication in 3GPP systems. 3GPP TR, Mar. 2007.

[20] UUSITALO, M. A. Global vision for the future wireless world from the wwrf. *Vehicular Technology Magazine, IEEE 1*, 2 (2006), 4 –8.

Between Neighbors: Neighbor Discovery Analysis in EH-IoTs

Shruti Devasenapathy*, Vijay S Rao*, R Venkatesha Prasad*, Ignas Niemegeers* and Abdur Rahim†
*Wireless and Mobile Communications,
Faculty of Electrical Engg., Mathematics and Computer Science,
Delft University of Technology,
Mekelweg 4, 2628CD Delft
The Netherlands
{S.Devasenapathy,V.Rao,R.R.VenkateshaPrasad,I.G.M.M.Niemegeers}@tudelft.nl
†CreateNet
abdur.rahim@create-net.org

***Abstract*—Devices in future Internet of Things (IoT) will be scavenging energy from the ambiance for all their operations. They face challenges in various aspects of network organization and operation due to the nature of ambient energy sources such as, solar insolation, vibration and motion. In this paper we analyze the classical two-way algorithm for neighbor discovery (ND) in an energy harvesting IoT. Through analysis, we outline the parameters that play an important role in ND performance such as node density, duty cycle, beamwidth and energy profile. We also provide simulation results to understand the impact of the energy storage element of energy harvesting devices in the ND process. We demonstrate that there exist trade-offs in choices for antenna beamwidth and node duty cycle, given node density and energy arrival rate. We show that the variations in energy availability impact ND performance. We also demonstrate that the right size of the storage buffer can smooth the effects of energy variability.**

I. Introduction

In today's connected world, we have several means of communicating with others on the go. Not only this, we can also connect to various objects in our surroundings for various services. We are on the verge of a future where there will be thousands of inanimate objects to each human that will ceaselessly communicate with each other to support humans. In general this paradigm is termed as "Internet of Things" (IoT). The nodes or devices in IoT are predominantly sensors and actuators that will work towards better home and office automation, maintenance and control of operations within systems such as vehicles, process industries, stock management at stores and industries, and so on.

With the growth of number of these devices that would be networked to support various activities, it is impossible to power them continuously through grid or batteries. One way to power them continuously is through harvesting energy from the ambiance. In light of the number of IoT devices that are predicted to be employed (25 billion by 2015 [1]) and the nature of applications, it becomes important to study devices that employ energy harvesting technologies as we elaborate in this paper.

The importance of neighbor discovery in IoT devices is evident when considering the nature of these devices and networks that need perform self-management, self-healing and self-configuration. An IoT device needs to learn about its immediate environment to be capable of performing these tasks, thus justifying a dedicated study towards neighbor discovery protocols in this special class of networked devices.

Energy harvesting depends on the ambient source. It depends on the deployed location and on the type of the harvester as well. Energy harvesting at any node varies heavily with space and time across the deployed set of IoT devices, leading to the necessity of designing communication protocols that accommodates these variations. Specifically, in this paper we focus on the issue of neighbor discovery (ND) for energy harvesting IoT (EH-IoTs) devices. In traditional sensor networks, ND is performed implicitly. Given the variable instantaneous energies in EH-IoTs, however, the energy harvesting nodes can leave and re-enter the network. Thus, ND is no longer a trivial task in such networks and also it is not only performed at the deployment stage of the network but also at regular intervals. In EH-IoT networks, every node may see different energy availability e.g., a device with photovoltaic (PV) panel facing south and another facing north. Such heterogeneity implies that the burden of ND could be handed over to a node that sees more frequent or larger quantities of energy. If nodes are equipped with prior knowledge of energy availability through a reliable energy prediction algorithm, they could pro-actively support ND process.

It is interesting to investigate with both directional and omnidirectional antennas for the ND process in EH-IoTs. In the directional case, intuitively, the nodes' responses for discovery messages may have lesser collisions due to lesser node density, and hence may take lesser time for discovery of all the nodes. Additionally, this would result in energy savings, further contributing to lower discovery time.

In this paper we propose a general analytical model for ND in an EH-IoT setting. We make notes on the impact of important parameters - beamwidth, duty cycle, density – on performance, a detailed study of the effect of energy variability and its mitigation through the choice of energy storage capacity. We do this with the help of the analytical model of a setup in which one EH node attempts ND to find its immediate neighbors (all EH-IoT devices) with an omnidirectional and directional antenna. Through numerical results from this model, we find the extent of impact of important parameters. These are supported with simulation results.

The organization of this paper is as follows. We introduce some pertinent related literature in Section II. We describe an ND algorithm that follows a simple three-way handshake for nodes to discover each other, as an analytical model in Section III. The parameters that influence ND performance are outlined with the help of numerical results in Section IV. The results of the simulation of this ND algorithm under a realistic energy regime are discussed in Section V. Numerical and simulation results are discussed and recommendations are provided in Section VI. Finally, Section VII concludes this paper with some recommendations for system and network choices.

II. Related Work

ND in wireless sensor networks is not considered to be a problem by itself as it is traditionally performed by MAC protocols as an implicit operation. However, as Dutta and others point out in [2], ND is not a trivial problem in networks where it is not easy or practical to predict if and when a node will find a neighbor nearby. These networks include mobile networks in which energy is a constraint – e.g., battery operated ad hoc networks. With respect to low or optimal energy consumption, Madan and Lall present a minimum energy method for ND [3]. Their solution rests on the computation of a minimum energy graph at design time by solving a stochastic shortest path problem. Both Dutta and Madan consider energy conservation in constrained setups during ND. However, the problem of energy conservation is not primary in energy harvesting. Over a long duration of time, a node can receive sufficient energy to perform various operations, but instantaneous availability is limited. Thus, the major issue posed by harvesting is variations in energy availability and thus must be handled differently. Furthermore, storage of energy in devices such as supercapacitors introduces loss due to leakage, which implies it is not ideal to wait for energy accumulation in storage devices for use over a long period of time. Thus, the design of a network of EH-IoTs is non-trivial.

Iyer et al., define a protocol *NetDetect* where neighbor discovery is performed using periodic beacon transmissions [4]. Here, the rate of beaconing is based on the estimate of the number of nodes in the neighborhood. However in case of energy harvesting networks, a popular technique adopted for energy management is to adapt the duty cycle to the rate of energy harvesting [5], [6]. Such a rate adaptation causes additional complexity in ND process.

Various ND algorithms have been analyzed for wireless networks that use directional antennas. Vasudevan et al. [7] classify them into two major categories – direct and gossip based. They present an analytical approach to comparing algorithms. The authors discuss antenna beamwidth selection based on the optimal transmission probability for best performance. Similarly, the effect of beamwidth and propagation models on the performance of ND protocols for 60 GHz networks is analyzed in detail by An et al. [8]. Analysis of ND in ad-hoc networks is discussed by An and Hekmat [9]. Several works deal with neighbor discovery in ad hoc networks [10], [3], [11]. Zhang and Li conduct a performance analysis of several random and scan-based algorithms for directional ND in ad hoc networks [12]. Their study concludes that an iterative scanning method performs better. We employ such an iterative scanning method in our study as well.

The impact of variations in energy availability on the ND process (whether omnidirectional or directional), as is observed in an energy harvesting system, has not been investigated to the best of our knowledge. In order to study this, it becomes important to understand the nature of energy harvesting sources. Energy arrival at a harvesting node in a natural environment is best modeled as a stochastic process due to the random nature of most natural sources such as sunlight and wind. Poggi and others demonstrate that the solar radiation recorded over a period of time can be mapped to a Markov process [13]. Similarly Ho et al. provide the methodology to model harvested energy as a non-stationary Markov process with added context - that is additional information about the environment in which the device is deployed[14]. Similarly, in our study, we model energy arrival as a stochastic process to emulate natural conditions. Thus, we shed some light on the impact of various important parameters and importantly varying energy availability on ND in an energy harvesting setup.

While there is a large body of work on the subject of neighbor discovery in ad-hoc networks and also in energy-constrained wireless networks, existing literatures do not sufficiently address ND specifically in energy harvesting networks. In this paper, we focus on the special circumstances of EH networks that warrant different solutions than those proposed before.

III. Analytical Model

In order to understand the factors that impact the ND process, we study the analytical model of the two-way ND process which a single node performs to discover all of its k neighbors. We assume that the number k is known a priori. Other assumptions are that each node can control the beamwidth of its antenna, all nodes operate at a duty cycle and each node is equipped with an energy harvester. Further, time is divided into equal slots each of which could refer to one millisecond (as we consider in our simulation study) for instance. The nodes are synchronized with respect to the slots i.e., two nodes waking up at a slot will wake up together. However, note that nodes choose their own wake-up and sleep times depending on their energy and duty cycle.

We consider a rectangular field in which nodes are placed at random. One of them is picked to be the "scanning node"

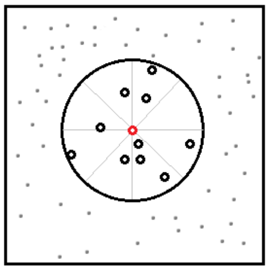


Fig. 1: Example Setting for ND in a set of EH-IoTs

A (marked in red in Fig 1) referred to as A. The scanning node is similar to a sink or a cluster head since it possesses higher processing capabilities. This scanning node attempts to discover all of its immediate neighbors $B_1, B_2, ..., B_k$ (marked as black circles) using the two-way ND process. We define immediate neighbors to be those nodes that fall into the radio range (given by the large circle) of the scanning node A. Further, the area that falls in the radio range of A is divided into sectors (as an example into 8 sectors marked with gray lines in Fig 1) that represent the area that A spans with its directional antenna.

Thus, we assume a hierarchical structure with a single node that acts pro-actively to initiate and perform a neighbor discovery process. This scenario can be expected, for example, in smart home settings where a single node behaves as the home controller or supernode. Also, even in a homogeneous setting, deployed nodes do not assume the role of a lead as soon as they come up. Instead, they look for a lead or cluster head and engage with it. Our assumption allows us to approach reality better than in a completely homogeneous model. Furthermore, this is an open problem for energy harvesting networks in general for the following reasons. Since adaptive duty cycling and varying energy conditions make it impossible for nodes to ensure the absence of other scanning nodes by simply listening long enough, nodes cannot arbitrarily choose to become scanning nodes themselves. An election process where nodes choose the node that becomes a cluster head is not possible before nodes become aware of their neighbors first, that is not before ND is performed.

The neighbor discovery process involves exchange of a set of handshake messages between the scanning node and its immediate neighbors. The scanning node initiates the process by sending out an ND packet. Any neighbor node that hears the ND packet responds to it with a response ND (RND) packet. If the scanning node receives the RND successfully, it sends an acknowledgment. If not, the neighbor node changes its next instant of wakeup and the process repeats. At the end of a single successful run of this process, the scanning node and a given neighbor have listed each other in their respective neighbor tables.

In this study, performance metrics considered are : ND time – the total time taken to discover all one-hop neighbor nodes and ND ratio – the ratio of number of found nodes to total number of one-hop neighbor nodes. We focus on ND time which provides an understanding of the efficiency of the ND process.

A. Omnidirectional Neighbor Discovery

First, we understand the behavior of the omnidirectional transmitter. A must discover k nodes labeled B_1 to B_k (we refer to a single one of them simply as B). These are k nodes that happen to be within the radio link range of A (marked as black circles in Fig 1). In order to discover these nodes, A transmits with a probability P_{t_A} given as,

$$P_{t_A} = \frac{1}{N_{2way}} P_{eA} \tag{1}$$

A attempts ND at regular intervals once every N_{2way} time slots. The parameter P_{eA} is the probability that A has the energy required to initiate and complete the ND process at that instant of time. This probability in a real system would be equal to the probability that the required amount of energy is available to node A. The parameter P_{t_A} would describe periodic discovery attempts if the device operated on a steady energy supply, e.g. mains supply. However, energy availability is random in an energy harvested device. Therefore, though the node is scheduled to perform discovery at regular intervals, the process can be best described as random due to the parameter P_{eA}.

Every node B listens for an ND message from A with a probability, P_{l_B} which is given as:

$$P_{l_B} = \frac{1}{T_B} P_{eB}, \tag{2}$$

where T_B is the on duration of B_i (the node is ON for 1 time slot) and P_{eB} is the probability that B has the energy to respond to the ND message. Again, for a device running on a constant energy source, node B would listen for ND messages at fixed regular intervals. However, the variability in energy availability at given instances causes the listening interval to best described as random. It is important to note here that the instances at which A and B may perform their respective activities are fixed and regular, whether or not they do perform scheduled activities at a given instant is dictated by the availability of energy. Also, N_{2way} and T_B are chosen to be co-prime, in accordance with the Chinese remainder theorem [15], so that there is never a possibility that two nodes do not discover each other.

Thus, the probability that an ND packet transmitted by A reaches B successfully is given as:

$$P_{A \to B} = P_{t_A} P_{l_B} \tag{3}$$

This also gives the probability P_{t_B} that node B that receives this ND packet responds to it by sending an RND. In order for the discovery process to be completed, B_k nodes must respond to A without their RNDs colliding with each other. The probability that of k nodes that have not been discovered by A of which only one responds is given as $(1 - P_{t_B})^{k-1}$. Thus the probability that only a single node reaches the scanning node successfully (without collisions [1]) is given as:

$$P_{B\to A} = \binom{k}{1} P_{t_B}(1-P_{t_B})^{k-1} \tag{4}$$

From this expression it is possible to calculate the time required to find a given node B_i as,

$$NDTime(i) = \frac{1}{P_{B_i\to A}} \tag{5}$$

The total time that is required to find all k nodes is given as,

$$NDTime = \sum_{i=1}^{k} \frac{1}{P_{B_i\to A}}, \tag{6}$$

where i denotes the number of nodes out of k that have been found by A.

The conflict resolution mechanism assumed here can be described in a practical setup as a three-way handshake mechanism. The scanning node transmits an ND packet, waits for an response (RND) and retransmits the received RND. Thus both nodes are informed of the success or failure of this exchange if the ND and RND packets are received correctly at both ends. In case of a failure to receive a correct RND, the neighbor node shifts the instant of its next wakeup by a random number of time slots. This helps avoid the inadvertent synchronizing of two or more neighbor nodes. These assumptions are not factored into our analysis, since we focus here on describing the scanning node's performance and the conflict resolution mechanism is implemented only into neighbor nodes.

B. Directional Neighbor Discovery

Let us now consider a scanning node whose beamwidth is controlled such that it transmits only to a small sector of its radio link area. There are changes in the analysis that we address in this section. We label the directional scanning node A_d. As in the omnidirectional case, the probability of transmission at A_d is given as :

$$P_{t_{A_d}} = \frac{1}{N_{2way}} P_{eA_d}, \tag{7}$$

and the probability that the neighbor node B is listening remains the same as before, given by Eq 2.

The probability that the node B is in the same sector as the scanning node is given as $\frac{\theta}{2\pi} = \frac{1}{N_s}$, where θ gives the beamwidth of the directional beam and N_s is the resultant number of sectors into which the circular field is divided. Thus the probability that node B responds to the ND packet from A_d is given as $P_{t_B}(1/N_s)$ where P_{t_B} is defined as before in the omnidirectional case. The probability that no other node responds to A_d depends on the number of nodes that fall within the beam of A_d. We define the probability that of k number of neighbor nodes there are j nodes in the same beam sector as our selected node B. In other words, the probability that the

[1]As we do not study the physical layer here, we assume that if an ND packet is transmitted without ND collisions, it can be received by devices in range with probability 1.

sector N_j containing node j is the same as sector N_B which contains node B is:

$$P_j = P[N_j = N_B] = \binom{N_s}{1}\binom{k-1}{j-1}\left(1-\frac{1}{N_s}\right)^{k-j}\left(\frac{1}{N_s}\right)^{j} \tag{8}$$

Finally, the probability that the node B is successfully discovered by node A_d is given as :

$$P_{B\to A_d} = \sum_{j=1}^{k} P_j \binom{j}{1} \frac{P_{t_B}}{N_s}\left(1-\frac{P_{t_B}}{N_s}\right)^{j-1}, \tag{9}$$

and this reduces to Eq 4 for $N_s = 1$. Again, the number of time slots required to discover a single node is given as $1/P_{B\to A_d}$ and the ND time for all k nodes is the summation for all nodes B_i :

$$NDTime = \sum_{i=1}^{k} \frac{1}{P_{B_i\to A_d}}, \tag{10}$$

where $i = 1, 2, ...k$ denotes the number of nodes found by node A_d.

IV. Numerical Results

From the analysis above, we can list the parameters that define the performance of the ND process as below: (a) Energy availability (P_{eA}, P_{eB}), (b) Number of nodes or node density (k), (c) Node duty cycle (given by P_{t_A} and P_{t_B} (d) Beamwidth (which gives N_s). In order to understand the impact of each of these parameters on the performance of A and A_d we plotted numerical results.

A. Energy Availability

In Fig 2, we see numerical results from the analysis for the case that $k = 40$ under several energy availability probabilities $P_e = P_{eA} = P_{eB}$ for all nodes. In our analytical model, we have not considered the dynamics of an energy storage element. Hence, the numerical results indicate the performance of an ON-OFF system. By considering the energy source to be random, we allow the model to be applicable with to a variety of harvesting sources that may or may not be time-dependent in nature, e.g. vibration-based or wind energy based energy sources.

The numerical results from this analysis for the case of varying energy probabilities and a scanning node that has a beamwidth of $\theta = 45°$ is given in Fig 3. With lower energy probabilities, the performance of ND deteriorates.

B. Node Density

The ND time for various values of k or node density for both an omnidirectional and directional transmitter is observed in Fig 4. As can be seen, the trend seems to saturate at higher values of k. This behavior is due to the fact that once a node has been found, it does not respond to subsequent ND packets of the scanning node. This factor i that takes values 1 through k, (appearing in Eq 6 and Eq 10) represents the number of nodes found at a given instant of time during the ND process .

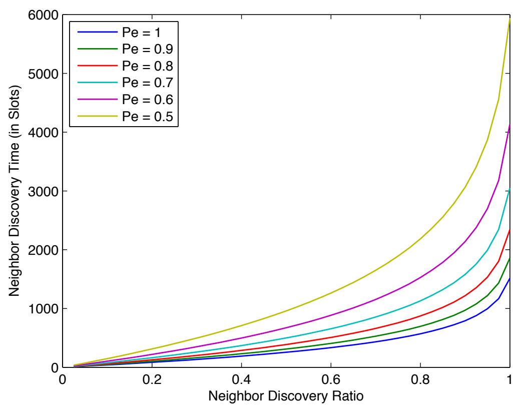


Fig. 2: Omnidirectional Two-Way ND for Various Energy Probabilities

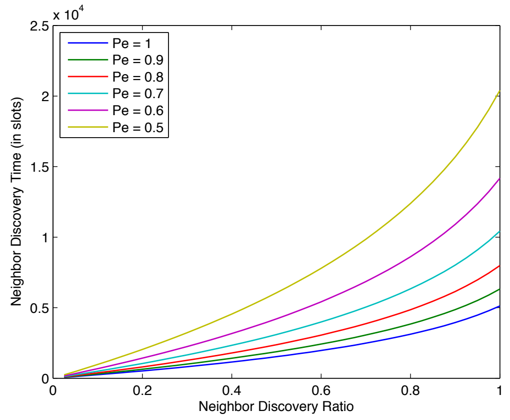


Fig. 3: Directional Two-Way ND for Various Energy Probabilities

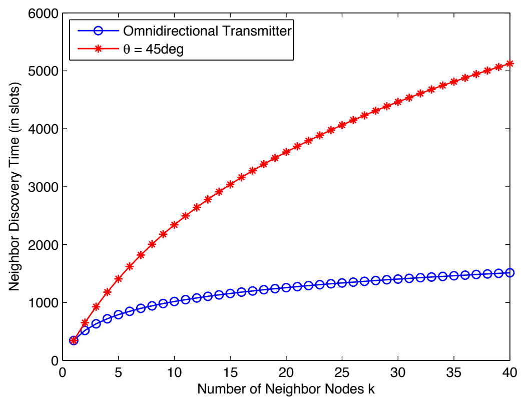
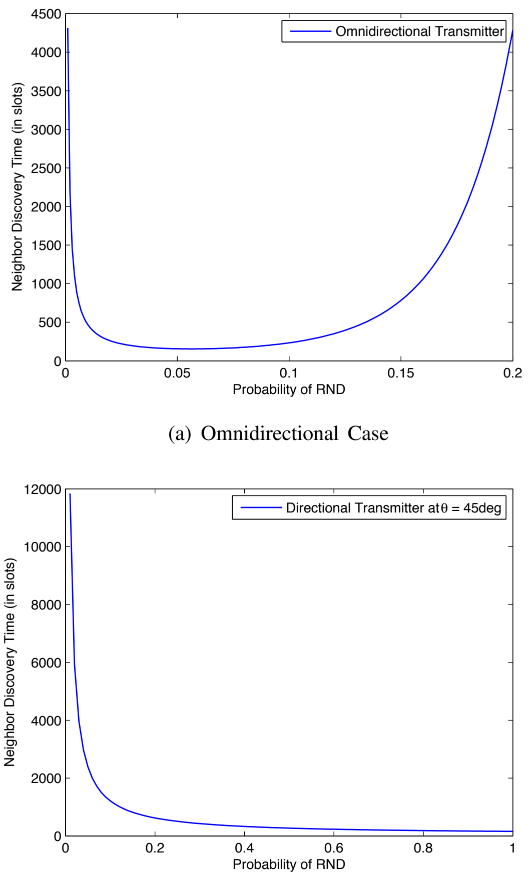


Fig. 4: Performance of Two-Way ND for Various Node Densities

C. Node Duty Cycle

The duty cycle gives the probability that a responding node B is awake and hence hears the message to respond to the ND packet from the scanning node. Thus, the parameter of interest here is the probability that a neighbor node responds to the ND given by Eq 4 for the omnidirectional case and by Eq 9 for the directional case. Plotting the ND time against this parameter gives us insight into the advantage that directional transmission would provide to the ND process.



(b) Directional Case

Fig. 5: Effect of Response Probability on ND Time

On the one hand, we see in Fig 5(a) that the ND time increases dramatically, starting at the extremely low RND probability of 0.12. On the other hand, the directional transmitter benefits from the factor P_j as there is little impact of a high probability of response (in Fig 5(b)). In other words, since there is a low probability of several of the k nodes occupying the same sector, the ND time is relatively unaffected by the response probability.

D. Beamwidth

We can see in Fig 6 that the performance gets progressively worse with an increasing beamwidth. The major cause for this is the lower probability that at a given time node B is listening in the same sector as the one in which A transmits. Another important factor is the variability of energy availability. We shall see the ameliorating effect of a storage buffer through simulation results.

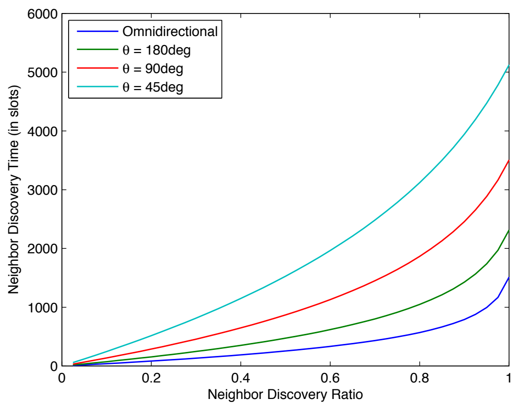


Fig. 6: Effect of Various Beamwidths on ND time

V. Simulation Results

We simulated the setup for the two-way ND scheme for various parameters. Presented in this section are the results of the simulation averaged over 1000 runs for each case. We introduce an energy storage element here which is modeled on a supercapacitor with linear leakage. Energy arrives to the supercapacitor following a Poisson arrival process, where each arrival is a uniform random variable which denotes the amount of energy harvested. We denote the inter-arrival times of this process as IA. Every node has an energy storage element and will initiate or respond to ND process if it has energy greater than a threshold value. If energy is unavailable, ND is deferred till the next scheduled time slot. This energy model [2] differs from the ON-OFF energy model considered in Section III.

The energy model considered in the simulation setup is specific to an energy harvesting setup that consists of a harvesting sensor (e.g. solar panel or thermoelectric generator) with harvesting electronics used to step up and regulate harvested energy and an energy storage device that has very specific properties. However, our analytical energy model is a much simpler ON-OFF setup with no storage capacity factored in. This simple model can be applied with small modifications to describe any other harvesting setup. While our motive here is to make a case for the necessity of ND algorithms for energy harvesting systems, an analytical model of a global energy harvesting model remains an open problem to be explored further [17].

A. *Effect of Energy Inter-arrival Times*

We see the impact of varying IA values on the ND time in Fig 7. When $IA = 15$, energy arrival is infrequent and the system suffers from greater variation in energy availability.

[2]Further details on the simulation energy model and results can be found in the companion paper [16].

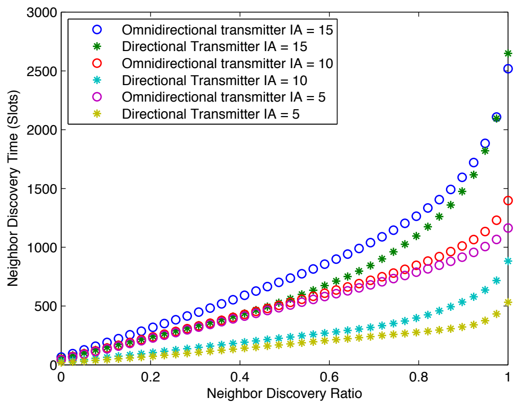


Fig. 7: Effect of Energy Interarrival Times for $k = 40$ on ND time

Under this condition, the omnidirectional scanning node outperforms the directional node. However, when energy availability improves, the directional node recovers substantially. We may attribute this behavior to three causes: (i) Lower energy availability impacts the directional node more than the omnidirectional node as we have observed from numerical results (ii) The directional node consumes more energy since it must transmit more ND packets. (iii) There is a smaller probability that neighbor nodes are listening in the same sector as the one in which the scanning node transmits – thus more energy is depleted from storage buffers and ND process gets deferred more often. However, it can be seen that the directional transmitter at $IA = 10$ performs even better than the omnidirectional transmitter that sees more frequent energy arrival at $IA = 5$. Thus we may conclude that the impact of energy variations on ND time is higher than that of beamwidth.

Though the energy model used to obtain numerical results in Section IV is different from the model in simulation, we see that the trends in Fig 7 match numerical results. This implies that our analytical model provides us with an upper bound for ND in energy harvesting systems, since they depict a scenario with no energy storage. We see that adding an energy storage buffer reduces the ND time considerably, which leads us to investigate the impact of size of storage element in the next subsection.

B. *Effect of Energy Storage Capacity*

In order to understand the impact of variations in energy availability during ND, we simulated the two-way process for different supercapacitor capacitance values 'C'. While the default value for C was $0.7F$ which corresponded to $3mJ$ of energy storage (for results seen in Fig 7), we see the ND time for various values of C in Fig 8. The study was conducted for the $IA = 15$ case to understand how the system behaves in the most adverse conditions.

As C increases, the ND time reduces. Since the storage buffer can store larger amounts of energy, the effects of constantly changing input energy conditions is reduced as the

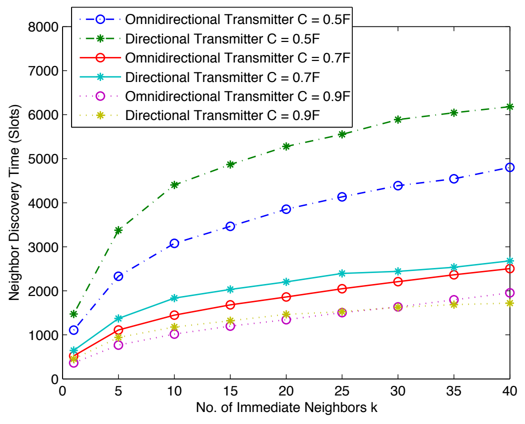


Fig. 8: Effect of Supercapacitor Capacity on ND time

supercapacitor now has a smoothing effect on these variations. Another interesting consequence of an increased C is that the difference in the ND time between the directional and omnidirectional scanning node reduces. For the case where C = $0.9F$, the directional scanning node matches the performance of the omnidirectional one for low values of k and even betters it for $k = 35$.

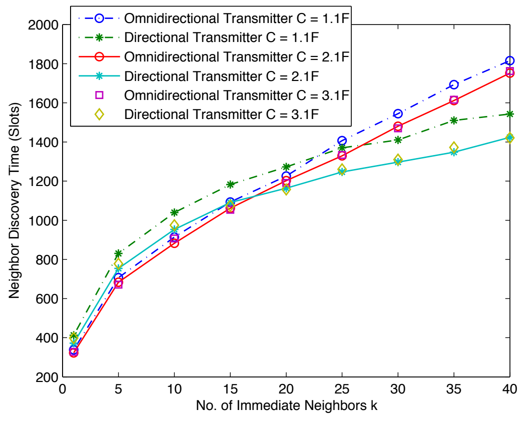


Fig. 9: ND Time does not improve after a threshold C

This improvement in performance is not seen beyond a threshold C. We see in Fig. 9 that there is no improvement in ND time from $2.1F$ to $3.1F$ and very little improvement from 1.1 to $2.1F$. It can be seen that the value of k at which the directional transmitter performs better than the omnidirectional one reduces from at $k = 35$ at C= $0.9F$, (seen in Fig. 8) to $k = 25$ at C = $1.1F$ and to $k = 20$ at C = $2.1F$. This allows us to conclude that energy variability does impact the directional transmitter more but these effects can be smoothed by using the right size of storage buffer.

VI. Discussions

We have seen that the response probability of neighbor nodes has a great impact on the ND time. Response probability of B is in turn heavily impacted by energy availability in a storage buffer. If this energy is wasted, for example on collisions, the effect on performance is high. Collisions and the associated heavy losses occur more often in an omnidirectional case than in a directional case due to the factor P_j (Eq 8).

We have seen the large effect of node density and duty cycle on performance. The choice of these parameters must be taken into account during design and deployment in order to achieve best performance at least cost to the entire network – given network parameters such as connectivity and redundancy. Since there is an obvious trade-off between the node density and the duty cycle of neighbor nodes, it is important to choose one parameter given the other.

Next, importance must be given to the impact of beamwidth with respect to number of neighbors. For example, if the number of neighbors is low, it is advisable to use a larger beamwidth in interest of energy expenditure.

Finally, our numerical results suggest that though the directional transmitter sees lesser impact of response probability, it suffers heavily due to the decreased probability that nodes A and B are in the same sector at the same time. Variations in energy availability across the network also impact the directional antenna. Nevertheless, we have seen that by making right design choices for the energy storage buffer size, this impact can be handled well. So, an energy model for a given application setting must be first created in order to understand the frequency of arrival of energy that can be expected in that setting, to define the size of the storage buffer at design-time.

VII. Conclusion & Future Work

We described the analytical model for ND in a network of EH-IoT devices. With the help of this model, we outlined the impact of various important parameters – node density, node duty cycle, beamwidth and energy availability – on the ND process. Through numerical and simulation results, we described the extent of influence of these parameters on ND. We demonstrated the trade-offs that need to be resolved for good ND performance – tradeoffs between (i) node density and duty cycle, (ii) node density and antenna beamwidth, (iii) energy availability, energy storage and beamwidth. Finally, recommendations on how to make choices such that these tradeoffs were resolved were provided.

Future work must focus on studying ND algorithms for a more homogeneous setup. This remains a complex, open problem for EH networks. In a universally applicable energy harvesting model that remains an open issue, a non-linear leakage model for supercapacitors must be implemented.

VIII. Acknowledgment

We thank iCore project. This article describes work undertaken in the context of the iCore project, *Internet Connected Objects for Reconfigurable Ecosystems* (http://www.iot-icore.eu/). iCore is an EU Integrated Project funded within the European 7th Framework Programme, contract number: 287708. The contents of this publication are the sole responsibility of iCore project and can in no way be taken to reflect the views of the European Union.

We also thank Go-Green project. This article describes work partially undertaken in the context of the Go-Green project (http://gogreen-project.nl/). Go-Green is sponsored by the Dutch Ministry of Economic Affairs, Agriculture and Innovation.

REFERENCES

[1] *The state of Broadband 2012: Achieving digital inclusion for all*. Broadband Commission (ITU), Sep. 2012.

[2] P. Dutta and D. Culler, "Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 71–84. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460420

[3] R. Madan and S. Lall, "An energy-optimal algorithm for neighbor discovery in wireless sensor networks," *Mob. Netw. Appl.*, vol. 11, no. 3, pp. 317–326, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s11036-006-5185-x

[4] V. Iyer, A. Pruteanu, and S. Dulman, "Netdetect: Neighborhood discovery in wireless networks using adaptive beacons," in *Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on*, oct. 2011, pp. 31 –40.

[5] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1274858.1274870

[6] C. Moser, L. Thiele, D. Brunelli, and L. Benini, "Adaptive power management in energy harvesting systems," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, april 2007, pp. 1 –6.

[7] S. Vasudevan, J. Kurose, and D. Towsley, "On neighbor discovery in wireless networks with directional antennas," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 4, march 2005, pp. 2502 – 2512 vol. 4.

[8] X. An, R. Prasad, and I. Niemegeers, "Impact of antenna pattern and link model on directional neighbor discovery in 60 ghz networks," *Wireless Communications, IEEE Transactions on*, vol. 10, no. 5, pp. 1435 –1447, may 2011.

[9] X. An and R. Hekmat, "Self-adaptive neighbor discovery in ad hoc networks with directional antennas," in *Mobile and Wireless Communications Summit, 2007. 16th IST*, july 2007, pp. 1 –5.

[10] R. Murawski, E. Felemban, E. Ekici, S. Park, S. Yoo, K. Lee, J. Park, and Z. H. Mir, "Neighbor discovery in wireless networks with sectored antennas," *Ad Hoc Networks*, vol. 10, no. 1, pp. 1 – 18, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870511000771

[11] G. Jakllari, W. Luo, and S. Krishnamurthy, "An integrated neighbor discovery and mac protocol for ad hoc networks using directional antennas," *Wireless Communications, IEEE Transactions on*, vol. 6, no. 3, pp. 1114 –1024, march 2007.

[12] Z. Zhang and B. Li, "Neighbor discovery in mobile ad hoc self-configuring networks with directional antennas: algorithms and comparisons," *Trans. Wireless. Comm.*, vol. 7, no. 5, pp. 1540–1549, May 2008. [Online]. Available: http://dx.doi.org/10.1109/TWC.2008.05908

[13] P. Poggi, G. Notton, M. Muselli, and A. Louche, "Stochastic study of hourly total solar radiation in corsica using a markov model," *International Journal of Climatology*, vol. 20, no. 14, pp. 1843–1860, 2000. [Online]. Available: http://dx.doi.org/10.1002/1097-0088(20001130)20:14¡1843::AID-JOC561¿3.0.CO;2-O

[14] C. K. Ho, P. D. Khoa, and P. C. Ming, "Markovian models for harvested energy in wireless communications," in *Communication Systems (ICCS), 2010 IEEE International Conference on*, Nov. 2010, pp. 311 –315.

[15] M. H. Niven I, Zuckerman H.S, *An Introduction to the Theory of Numbers*. John Wiley & Sons, 1991.

[16] S. Devasenapathy, R. V. Prasad, V. S. Rao, and I. Niemegeers, "Impact of antenna directionality and energy harvesting rate on neighbor discovery in eh-iots," in *Proceedings of the 10th Annual IEEE-CCNC Smart Spaces and Sensor Networks*, Las Vegas, NV, USA, 2013.

[17] R. V. Prasad, S. Devasenapathy, V. S. Rao, and J. Vazifehdan, "Reincarnation over the air: Energy harvesting devices and networks," *IEEE Communications Surveys and Tutorials*, 2013.

Towards a generic architecture and methodology for multi-goal, highly-distributed and dynamic autonomic systems

Sylvain Frey[1,2], Ada Diaconescu[2], David Menga[1] and Isabelle Demeure[2]
[1]*ICAME department, EDF R&D, Clamart, France; {first name}.{last name}@edf.fr*
[2]*Télécom ParisTech, CNRS LTCI, Paris, France; {first name}.{last name}@telecom-paristech.fr*

Abstract

Autonomic control is vital to the success of large-scale distributed and open IoT systems, which must simultaneously cater for the interests of several parties. However, developing and maintaining autonomic controllers is highly difficult and costly. To illustrate this problem, this paper considers a system that could be deployed in the future, integrating smart homes within a smart micro-grid. The paper addresses this problem from a Software Engineering perspective, building on the authors' experience with devising autonomic systems and including recent work on integration design patterns. The contribution focuses on a generic architecture for multi-goal, adaptable and open autonomic systems, exemplified via the development of a concrete autonomic application for the smart micro-grid. Our long-term goal is to progressively identify and develop reusable artefacts, such as paradigms, models and frameworks for helping the development of autonomic applications, which are vital for reaching the full potential of IoT systems.

1 Introduction

The purpose of any computing system is to reach objectives specified by an external authority. When multiple authorities can access the system, like in the IoT (Internet of Things) context, system goals may be conflicting, while targeting overlapping system parts. Moreover, such systems must often scale to large numbers of highly-distributed resources and be adaptable to changes in their goals, execution context and constituent resources (the systems are open). Autonomic or self-* capabilities become key to the success of such systems.

This paper illustrates this challenge via a multi-goal, adaptable and open autonomic system that integrates several smart houses into a smart micro-grid. To cover both the Autonomic Computing (AC) and IoT domains in this example, the paper employs the generic term *autonomic control* [28] to designate the system logic that manages available resources for attaining goals. The only means for an autonomic controller to pursue its objectives is via actions it can perform on such manageable resources. To select actions the controller can rely on decision strategies, knowledge and runtime information from the environment and the system state. The **key challenge** lies in developing the controller logic that can successfully pursue system goals while ensuring essential system characteristics - scalability, robustness, adaptability and openness. We approach this challenge from a Software Engineering (SE) perspective. Our aim is to identify, specify and develop reusable artefacts for analysing and designing autonomic control systems with the aforementioned properties. The presented work relies on our experience with building autonomic frameworks [9][10][20][23]. The long-term aim is to build a comprehensive **reference architecture for autonomic systems**.

The generic architecture proposed is constructed on the assumption that the development and adaptation of any realistic autonomic system will rely on the *integration* of managed resources and control elements of different types; integration can occur statically or dynamically. An important challenge lies in identifying and bringing together the necessary types of *abstract architectural artefacts* and concrete *control elements* that can be used for system design and integration. Abstract artefacts can include architectural styles, design patterns and layering techniques over several axes of abstraction. Control elements include relatively straightforward control tasks - such as monitoring, decision-making, execution or knowledge-management; entire control loops; or combinations of the above [15][23]. They can be functionally organised based on well-defined abstract entities, like those indicated above, and interconnected via hard-coded or loosely-coupled bindings. The overall integration process can be controlled in a fully centralised, decentralised or hierarchical manner [15][9][10][14][23].

Another important challenge lies in coordinating control elements for obtaining coherent controllers that can pursue several goals, adapt and support highly-distributed, plug-and-play resources. Of major interest here is the detection and resolution of *conflicts* that may occur when integrating elements with contradicting goals [10] or control strategies [23]. The generic architecture and methodology presented here focus on addressing these two major challenges. Other important concerns, such as timing and synchronisation of integrated actions are part of ongoing research not covered here. The authors do not claim the novelty of all artefacts in the architecture. Indeed, most of these can be found in related fields such as automatic control [24], collective adaptive systems [18], multi-agents [14][16], robotics [6], cybernetics [2] or autonomic systems [15][7][19]. These provide a rich repertoire of solutions that address different parts of the overall challenge.

This paper's contribution consists in identifying and extending existing artefacts that can be used for designing autonomic control systems, and assimilating them into a coherent framework. Some **key aspects of the proposed contribution** include: rendering explicit the conceptual elements included in goal definitions; defining the problem of building autonomic controllers as one of mapping declarative actions (goals) into concrete actions (on managed resources), in a context-aware and extensible way; combining existing SE techniques for splitting the mapping problem into recursively smaller elements and integrating such elements into flexible overall solutions; defining integration conflicts and ways of resolving them; applying architectural templates and agent organisation techniques to ensure system coherence and runtime flexibility. This is illustrated by developing a multi-goal, adaptable and open smart micro-grid.

The ongoing aim is to help answer questions on:

- How to develop scalable and adaptable feedback loops?
- How to integrate multiple feedback loops for pursuing many goals at different scales?
- How to deal with system dynamism and openness?

Addressing these concerns is vital for reaching the full potential of Autonomic Computing and IoT paradigms. We emphasise the fact that we do not propose a concrete ready-to-use architecture; this would have to be domain-specific. Rather, we provide an abstract architecture and methodology that can guide the design process of autonomic control systems in various domains. The proposed contribution is relevant to both autonomic systems in general - as it helps design multi-goal, distributed and adaptive autonomic managers; and to IoT systems - as it shows how autonomic controllers built in this way can control system resources to ensure required properties and functions.

Section 2 describes the sample smart micro-grid application, with its requirements and design challenges. Sections 3 and 4 introduce the conceptual and design aspects of the proposed architecture, respectively, illustrating them via concrete examples from the smart micro-grid. Section 5 illustrates the complete design of the prototype application. Section 6 discusses related work and section 7 concludes the paper and indicates future research.

2 Smart Houses meet Smart Micro-Grid

2.1 Overall system

In a near-future, it can be envisaged that smart homes integrate with smart grids to form large-scale, highly-distributed, dynamic and open IoT systems. This paper considers this type of system as a relevant use case for the problem addressed. For the sake of clarity and expressiveness, the system model is often kept simple, neglecting important aspects such as business models, legal regulations or fine-grain grid behaviour.

Smart homes are seen here as cyber-physical systems that integrate and control electrical devices in order to provide automated services, such as context-aware heating, entertainment, lighting and security. Individual devices termed "smart" embed their own control logic to offer some service. For instance, a thermostat can turn itself up when detecting the home owner's presence.

A *micro-grid* is a local, low-tension electrical network. For simplicity, this paper considers a residential district organised as a tree, rooted at the district aggregator; the leaves are the end-user appliances - producers (e.g. solar panels), consumers (e.g. electrical heaters) or both (e.g. batteries). The generic term *prosumer* designates such endpoints; the associated term *prosumption* means either production or consumption. A residential tree is part of a city grid that is in turn part of the national grid (not considered here). A house grid is a sub-tree of the district grid. Its prosumption is measured by a house meter and equal to the sum of prosumptions of all appliances in the house. Likewise, the district's prosumption is the sum of all household prosumptions.

The *load* of a grid is defined as the ratio between productions and consumptions. It is said to be *high* when consumptions overshoot productions, hence requiring consumption from the parent grid; *low* load denotes the opposite. In this paper, load management consists in adjusting local productions and consumptions to minimise the footprint on the parent grid. For simplicity, the paper will globally refer to the "smart micro-grid" including implicitly the integrated smart houses.

2.2 Autonomic control requirements

Let us now define the perimeter of the smart grid's autonomic controller and identify its most important requirements. First, the controller must pursue several goals, specified by different authorities. The electricity provider imposes load management goals for the grid. In the presented scenarios, these goals take the form of a Goal Power (GP) interval - $[GP_{low}, GP_{high}]$ - within which the prosumption of a sub-grid should be maintained. The exact values will depend on business objectives at different grid scales and on the context. Home owners define different types of goals for their households. These may be related to comfort - like maintaining a temperature (heaters) or a lighting ambiance (lamps), or simply performing activities like washing (washing machine) or cooking (oven). They may also be related to cost - like minimising the electricity bill or the environmental impact. Note that such goals can be in conflict.

Hence, the autonomic controller must be able to either favour one goal over all others - like prioritising energy savings over appliance usage or conversely pursuing comfort at any cost; or target a compromise among all goals - like only ensuring comfort partially if the grid is highly loaded. Such preferences are specified by administrative authorities and may be context dependent (e.g. user presence or weather). Finally, some preferences can be overridden implicitly as users handle appliances directly (e.g. turning-up a heater or cooking).

The autonomic controller must pursue its goals by performing actions on manageable resources, including grid resources (not discussed here) and electric appliances. The presented use case focuses on two sample appliances with specific profiles. First, heaters transform electric energy into heat; their power can be monitored and set via specific touchpoints. Second, lamps transform electric energy into light; their light intensity can be adjusted via specific touchpoints that measure and set their consumption. While lamps do not usually constitute significant consumers, they are used here to model diverse equipments with similar profiles, such as microwave ovens or vacuum cleaners. Finally, privacy concerns impose that house appliances cannot be controlled from outside the house within which they reside.

In addition to meeting the goals, the autonomic controller must scale to large numbers of highly-distributed resources (e.g. appliances). Also, the controller must adapt to changes in goal specifications (e.g. power intervals), priorities (e.g. comfort vs. savings) and execution context (e.g. weather). Finally, it must handle "smart" or standard appliances being plugged-in or out.

3 Conceptual Model

3.1 Goal types and specifications

Goals represent the very purpose of autonomic systems. Generally, they define a system's *viability zone*, within which its state must be included at any one time [15][1][2]. A system's state is defined via a set of variables whose values can predict its behaviour in the near future [24] (e.g. a heater's power setting predicts the amount of heat it will produce). A system's state can also represent its end goal (e.g. a targeted temperature). Goal definitions are intimately related to the way in which they can be evaluated - typically via observations on system state variables. Goals may be declarative or procedural [17]. Declarative goals indicate *what* should be achieved rather than *how*. They are usually defined as constraints on system variables, delimiting the viability zone, and can be evaluated automatically via a utility function over the system state. Procedural goals indicate (via high-level policies) *how* the system should behave in various situations. This paper focuses on declarative goals, considering that procedural goals can be induced from these.

A goal definition can include three types of elements - **G (V, S, T)**, where V defines the viability zone, S the resources to which it applies, and T the periods over which it applies. The viability constraints (V) are compulsory and typically accompanied by a utility function for evaluation purposes. In the smart home example, a goal can define a viability interval for the power consumption. The Scope element (S) separates the viability definition from the resource domain to which it is applied (and evaluated). It is defined via domain constraints that identify, in a declarative way, the system resources targeted at any one time. In cyber-physical systems, such as IoT, Scopes can represent physical areas in Euclidian space; resources located in that area belong to the Scope. For instance, a goal defining a temperature interval can be applied to the scope of a house or only of one room. It will be evaluated using thermometers located across the house or within that room, respectively. In systems where physical space is less relevant Scopes can define other types of resource sets - e.g. a network domain in a computer cluster. Scopes are particularly relevant to open systems, where resources can change dynamically and unpredictably. For example, a power interval can be defined for a house, without explicitly identifying all its appliances. Finally, the Time element (T) separates a goal's viability constraints and scope from the periods over which they take effect. For simplicity, this element is no further developed in the paper; goals implicitly start when received and end when cancelled or overwritten.

3.2 Goal achievement and evaluation

The only means for an autonomic system to attain its goal(s) is via actions it can perform on manageable resources [Fig. 1]. Namely, an autonomic controller should act so as to influence the variables of resources within the goal's scope (S_G) to maintain them within the goal's viability zone (V) - e.g. to pursue a temperature goal in a home, a controller acts on the heaters available in that home. It can be noted here that the set of resources on which the controller acts - *action resources* - is not necessarily equal to the set of resources in the goal's scope - *goal resources*. The only constraints are that the controller should be able to monitor goal resources for evaluating its goal; and that the action resources should have a controllable influence on the state of goal resources. Considering a temperature goal in one room: the room's atmosphere is the goal resource, since its temperature is monitred and evaluated; heaters in the room and in neighbouring rooms are action resources, since the controller acts upon them to *influence* the room temperature. A controller's action resources for pursuing a goal constitute its *Action Scope (S_A)*. The set of resources whose state they influence is referred to as *Influence Scope (S_I)*. Finally, the controller may monitor resources it cannot control - *context resources* from a *Context Scope* - e.g. outdoor thermometers. In summary, a controller pursuing a goal $G(V, S_G)$ will act on resources in an action scope S_A to influence resources in a scope S_I, where S_I includes the resources in S_G that are monitored to evaluate the goal [Fig. 1].

This approach clearly separates a goal's definition from the controller's means to pursue it. This is vital for adapting a controller's strategy to changes in its goals, environment and internal resources. It can also intervene in tackling multi-goal conflicts, as discussed later. This conceptual setting allows formulating the problem that an autonomic controller must solve - i.e. how to attain its goal(s). It consists in finding a strategy, or *mapping function*, which can **transform goals into concrete actions**; the solution will *be sensitive to the external context and internal system state*. This view generalises the notion of *goal to represent a higher-level declarative action* (intentional) that must be translated into *concrete actions* (A), executed via resource effectors (imperative) [15].

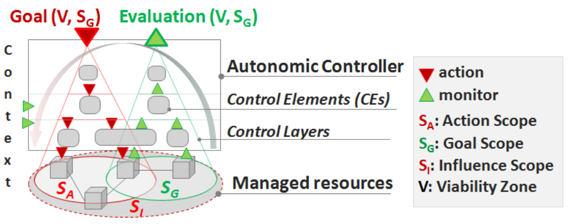


Figure 1: Goal projection and evaluation.

3.3 Goal translation and division

This subsection identifies the main factors behind the difficulty of mapping goals into concrete actions and indicates the structural and behavioural concepts that can help analyse and address them. **One factor** stems from an increasing "distance" - or difference in the *abstraction levels* - between the goal's viability specification (V) and the concrete actions (A). E.g., in the smart house, a controller must map a "comfort" goal into concrete power configurations on heaters. **A second factor** represents a typical control problem, involving complicated *decision-making* capabilities that rely on partial knowledge, react to fluctuating inputs, avoid oscillations and optimise results. **A third factor** intervenes as controllers must *adapt* to change - e.g. integrate plug-and-play resources and change strategies to achieve evolving goals in a variable environment. **The fourth factor** stems from the *scale* of goal scopes (S_G). A large-scale S_G often implies a comparably large-scale S_A which is difficult to control, especially in an open context. This difficulty increases when plug-and-play resources are heterogeneous and belong to different legal authorities.

"Classic" Software Engineering (SE) techniques can be applied to help address these factors. **Layering** can structure controllers along three distinctive axes. First, *abstraction layers* can progressively translate goals into concrete actions (abstraction factor). Each layer maps higher-level goals (or actions) from the layer above into lower-level goals (or actions) for the layer below [Fig. 1]; this results in a *translation hierarchy*. E.g., a "comfort" goal is translated into an intermediary "temperature" goal and then into a concrete "power" configuration. This controller has two abstract layers: the highest layer pursues the "comfort" goal (declarative); it acts by setting a "temperature" goal (declarative) on the lower layer, which acts by setting a "power" goal (imperative) on a heater. Goal evaluation follows the inverse translation path - monitored data from S_G resources (e.g. temperature) is translated into concepts of the administrative domain (e.g. comfort). Abstract layers are said to implement *base-level* functions, meaning to pursue goals by acting on resources. Second, *control layers* can be introduced to add meta-control abilities to such base-level functions, hence enabling controllers to self-adapt (adaptation factor). E.g., when a smart house's goal changes from "comfort" to "saving" mode, a meta-control layer can adapt the behaviour of control elements in the base layer, as necessary to pursue the new goal. Third, *integration layers* can be added to form a control hierarchy that helps integrate decentralised control elements (belonging to either abstract or control layers, as discussed below). As integration and abstract layers are often superposed in real system designs the terms are at times interchanged in the paper.

Encapsulation and modularisation techniques can complement layering to address a controller's complexity and adaptability concerns (decision-making and adaptation factors). They enable the separation of concerns in the controller's logic, facilitating the reuse and integration of simpler control elements (CEs) into complicated controllers. This is the equivalent of *splitting* the controller's mapping function into complementary parts. Adaptation can be achieved by replacing or reintegrating these parts. Domain-specific algorithms are necessary for implementing CEs and are outside the paper's focus. This technique can also be applied to split the goal's scope (scalability factor). Here, a goal (G) is split into complementary goals (G_i) that define the same type of viability constraints (V) over smaller scopes (S_{Gi}). Each G_i is assigned to a different control element CE_i. E.g., a comfort goal for a house is split into comfort goals for individual smart devices; or, the power goal over a district grid is split into power goals for different houses - here, the goal value for the district power constraint is also split into smaller values for each house grid. This approach can also address the multi-authority issue. E.g., district controllers (owned by a provider) split their goals among house controllers (owned by private parties). It also intervenes in goal translation to address resource heterogeneity. E.g., comfort is converted into temperature for thermostats, and into light intensity for lamps. **Loose-coupling and dynamic binding** enable runtime integration of CEs into adaptable controllers.

From a behavioural perspective, most CEs in the aforementioned structures act only in response to incoming data, like monitoring, analysis or action, from resources, other CEs or administrators. In an integrated system, CEs trigger each others' executions thus generating a *control flow* through the system [Fig. 2]. The control flow can pass through CEs within a single layer, like the MAPE elements of a control loop; as well as across layers, like a base-control loop triggering a meta-control loop or an integrator. This is an important concept and plays a key role in identifying and resolving conflicts. When a controller pursues a goal, we say that its control flow *serves* the goal or *carries* the ensued action(s).

3.4 Multi-goals, conflicts and resolution

Most autonomic systems will have to follow multiple goals, given by one or several authorities. In some cases, multiple authorities issue goals with the same type of viability constraints (e.g. range of power values) but with different constraint values (e.g. [1 kW, 2 kW] and [1.5 kW, 3 kW]). In another case, a single authority issues goals with different constraint types (e.g. comfort and power savings). The two cases can be combined.

Each goal can be addressed individually as discussed before. The solutions can then be combined to obtain multi-goal systems. The main additional problem intervenes when the system's goals are in *conflict*. This concept must be defined before addressing the problem. At the lowest system level, a conflict occurs when concrete actions attempt to change a resource's variables to incompatible values - e.g. one action turns a heater's power up and another one down. In most cases, conflict causes can be traced through the system to various sources. Source causes can stem from conflicting goals, conflicting controller strategies, or both of the above. Goals are conflicting when they define contradictory viability constraints over overlapping goal scopes (e.g. different power intervals over the same house grid). Control strategies are conflicting when they carry contradictory actions through overlapping influence scopes (S_I) (e.g. openning a window during a cold evening to pursue an air-freshness goal influences the room temperature, causing heaters to power-up and hence jeopertise a power-saving goal). Hence, conflicts *may* occur when goals can cause contradictory actions on overlapping S_Is, the intersection area being referred to as *Conflict Zone* [Fig. 2]. Concretely, conflicts *do* occur when control flows that service contradictory goals (or carry contradictory actions) pass through a conflict zone (within a certain period, which is not discussed here). To avoid such behaviour, conflict zones must be identified and special-purpose mechanisms placed in the CEs within those zones. These include conflict-resolution design patterns [10] or agent-like CEs that can compromise among goals (subsection 5.1). Several of these can be placed along conflicting control flows to improve the robustness of the resolution process [Fig. 2].

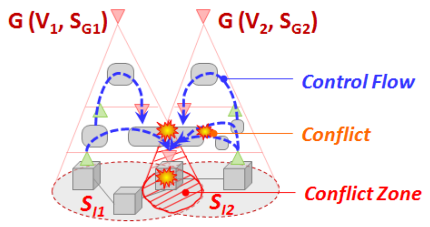


Figure 2: Conflicts and resolution.

4 Generic Architecture

4.1 Architecture Overview

We can rely on the abstract concepts discussed to define a generic architecture for designing autonomic control systems. To remain generic, the proposed architecture is a logical one, relying on and refining the Autonomic Computing Blueprint [15]. Here, an autonomic system consists of: *managed resources*, which can be acted upon and monitored; and an *autonomic controller*, which receives and pursues goals. We will show how controllers can be designed based on the layering, modularisation and loose-coupling techniques. The controller's abstract architecture will be customised and instantiated case-by-case resulting in application-specific designs. This process is highlighted in subsection 4.5 - 'methodology'.

Typically, an autonomic controller is divided into application-specific **abstract layers**, to bridge the conceptual gap between administrative goals and managed resource parameters. Each abstract layer can be enriched with one or several **control layers**, depending on the adaptability and goal-management needs of its control function (discussed below). **Modularisation** is introduced by splitting layers (abstract or control) into *control elements* (CEs) of various types (discussed below).

Loose-coupling enables the flexible *integration* of control elements (CEs) both within and across layers. Integration can be performed statically or dynamically, to initially develop and then adapt the controller. Hence, **integration layers** can be added to coordinate the disparate actions of CEs within abstract or control layers. Integration layers can also be modularised via CEs. To complicate things, integration and control layers can be further split into abstract layers and/or augumented with additional control layers.

In a generalised view, abstract, control and integration layers are conceptually orthogonal. In principle, any layer type can be divided recursively or added on top of layers of any other type. All layers are composed of CEs that can be added, updated or removed during runtime. In reality, particular combinations of layers and CEs types will most likely emerge in various application domains.

4.2 Types of layers and control elements

More concretely, CEs can represent: i) *control tasks* - control-related functions (e.g. monitoring, decision, execution, knowledge management, other atomic functions or combinations of these, as shown in [23]); ii) *integration tasks* - integration-specific functions (e.g. application-specific conflict resolution, as detailed in [10]); and iii) *control composites* - flexible compositions of control tasks and (optionally) integration tasks, for providing more advanced control structures and functions (e.g. single or integrated feedback loops). Control composites can or not be *encapsulated*. When encapsulated, they allow building fractal-like structures, which appear from the outside as a single well-integrated CE [Fig. 3], hence identical to a control task.

For instance, a base abstract layer that controls a heater to reach a temperature may consist of four control tasks: monitoring the temperature, analysing it with respect to a target, planning a power ajustement and executing it on the heater. If alternative planning tasks are available, an integration task can be added to select the best plan to use in each context. The integrated tasks form a complete control loop that can represent a control composite (encapsulated or not). This composite can be integrated with similar composites controlling other heaters and coordinated via an additional integration task.

From a behavioural perspective, CEs may be:

- *reactive* - acting in response to external stimuli; reactions can be stateless or using internal knowledge;
- *self-adaptive* - able to modify its reactions in response to changes;
- *agent*[1]*-like* - managing and negotiating goals given by other entities; only accepted goals are pursued.

To achieve such behaviours, the proposed architecture defines **three types of control layers**. Each CE may include one or several of these layers, in order to display a more-or-less sophisticated type of behaviour. First, a *base control layer* monitors and acts on managed resources following a pre-selected control strategy; it enables reactive behaviour. Second, a *meta-control* or *adaptation layer* ensures the base layer's adaptation to change, by altering or fine-tuning its control strategy [2][19]; it enables self-adaptive behaviours. Finally, the *goal management layer* receives requests for pursuing goals and decides whether or not to accept them; it enables agent-like behaviours, which are critical for conflict resolution. An agent's decision may be binary or more nuanced, based on the new goal's requester authority and conflicts with already accepted goals.

As indicated above, **abstract layers** for goal translation are application-specific - hence, no generic types were identified. Concerning **integration solutions**, several applicable designs were identified in [10] in the form of integration design patterns (e.g. hierarchy, controller, stigmergy or cooperation). While abstract, control and integration layers can be separated conceptually, in reality they can often be implemented within the same application layers of CEs (exemplified in section 5).

4.3 Requirements for integration

Integrating CEs must rely on standardised interfaces and protocols. While the details of these are domain-specific, their general semantics and purpose can be identified. This view is compliant with the Autonomic Computing Blueprint [15], but extended from control loops to all CEs [Fig. 3]. Hence, from an external view, CEs are quite similar. They *require* monitoring and action interfaces for accessing managed resources, including CEs in lower layers. They also *provide* monitoring and action interfaces for giving access to administrators and CEs in higher layers. These interfaces are the main enablers for CE layering and integration. Their semantics will differ depending on the CE type and conceptual layer - e.g., monitoring and execution touchpoints for control tasks in a base-control layer; and, goal specification and evaluation touchpoints for control loops in an adaptation layer. Their implementations will also differ - e.g., reactive CEs simply execute incoming actions, while agent-like CEs may execute, negotiate, or ignore them.

CEs may also provide and require functional interfaces for exchanges with other CEs [Fig. 3]. As before, these exchanges are application-specific, but their general purpose will depend on the CE's function - e.g. in the base-control layer, they can enable the integration of control tasks into feedback loops; in the self-adaptive layer, they can provide access to search and discovery services; for agent-like CEs, they can intervene in agent negotiation and self-organisation. Depending on its use, a CE may or may not provide all of these interfaces.

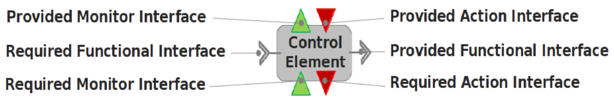

Provided Monitor Interface
Required Functional Interface
Required Monitor Interface
Control Element
Provided Action Interface
Provided Functional Interface
Required Action Interface

Figure 3: Control Element interfaces.

4.4 Integration and adaptation

Integrating CEs into multi-goal, distributed and adaptable autonomic controllers requires handling problems of communication, coordination and control. The proposed architecture identifies several integration-specific CEs to help with such issues. Essential elements include distributed communication infrastructures, discovery and repository services. Many such artefacts are available from related research domains and so not treated here. Conversely, coordination and control are key concerns that are especially challenging when CEs and resources must be integrated dynamically.

Two **complementary techniques** can be adopted to address these key issues. The first one relies on facilities for runtime evaluation, reporting and autonomic adaptation. These allow an autonomic controller to evaluate itself and adjust its internal composition accordingly, so as to remain within the viability zone. Support for this aspect was included in the conceptual model and will be further developed in future work. The second technique (developed here) relies on imposing architectural templates or *organisations* - a term borrowed from the agent community [14][26][29]. An organisation defines an *invariant system core*, or *template*, which can be "filled-in" dynamically with concrete resources depending on their availability and state. Imposing an organisation can ensure, to a certain extent, structural and behavioural properties for the resulting adaptive system [9].

An organisation consists of several *roles* that interact in a well-defined way. A role is defined as a set of well-specified capabilities - e.g., a "prosumer" role in a smart grid organisation. The role can be assigned (statically or dynamically) to any concrete resource that provides those capabilities - e.g., a heater takes the prosumer role. Based on the proposed architecture, autonomic controllers are designed as one or several organisations composed. Each organisation is defined based on the generic artefacts in the architecture, including layers and CEs. The smart micro-grid exemplified below shows how organisations designed in this way facilitate system development and adaptation. A catalogue of reusable organisations can be progressively developed. A core set was presented in [10] as integration design patterns. These include centralised orchestrators, decentralised coordination via functions embedded in CEs, hierarchical multi-layer organisations, aggregators and filter interceptors for integrated control flows.

4.5 Development methodology

We propose a certain sequence of *indicative steps* for developing autonomic control systems based on the generic architecture proposed:

1. specify system **goals**, defining their viability constraints and scopes - $G_i(V_i, S_{Gi})$; the authorities involved are also identified here.

2. identify the managed resources and the concrete actions that can be executed on them to influence the state of resources in S; these will make-up the controller's **Action Scope** (S_{Ai}).

3. identify 'relevant' resources that fall into the **Influence Scope** (S_{Ii}) of this Action Scope (S_{Ai}).

4. identify the context resources that can be monitored and influence control decisions; these will form the controller's **Context Scope** (S_{Ci}).

5. for each goal, define the main **abstract layers** and their control elements (CEs) required goal translation and splitting.

6. define an initial **integration infrastructure**, based on one or several **organisations**, ideally predefined in a catalogue. Essential services like communication and dynamic discovery must also be addressed here.

7. identify **conflict zones** (at each layer) based on the intersection of influence scopes (S_{Ii}) serving incompatible goals.

8. modify or extend the initial organisations with **conflict-resolution** facilities. One option (developed in the experiments below) is to add goal-management control layers to CEs located in the conflict zones; these CEs become 'agents'. Another option is to introduce special-purpose CEs implementing more complicated conflict-resolution design patterns (as shown in [10]).

9. enhance CEs (as needed) with self-adapting capabilities, by adding **adaptation-control layers** to them.

10. further **refine** any of the existing layers in any of the CEs (as needed) to enable more complex behaviours - e.g. add an adaptation layer to a CE's goal-management layer to change its policies during runtime; or, further split an adaptation layer into abstract layers to deal with its complexity.

The first four steps allow defining the controller's overall perimeter - its external inputs (goals and monitoring information) and outputs (actions and state reporting). This paper only focuses on goals and actions; future work will include evaluation and reporting. The fourth step starts defining the controller's internal design, based on successive layers and CEs. Steps five to seven set in place the integration infrastructure including conflict-resolution support. The last two steps refine the design with self-adaptation capabilities at all layers. If needed, they can further complexify each layer by splitting or enhancing them with otrthogonal layers. The smart micro-grid application exemplifies this procedure next.

5 Illustration via a smart micro-grid

5.1 Design and implementation

Like most SE contributions, evaluating the generic architecture proposed cannot rely on formal proofs and would require too vast experimental resources to rely on a meaningful empirical approach - e.g. [16]. Hence, for now, it can only be validated through rigorous argumentation and relevant exemplification, which is the aim of this section. We show how a smart micro-grid prototype was designed following the methology depicted above. Please note that many of the conceptual layers identified are superposed in the concrete controller design.

We first identify the **authorities** involved and the types of **goals** they could specify (step 1). The authorities are *electricity providers* and *home owners*. Electricity providers define *power goals* over their district grids - $G_{power}([P_{d\_low}, P_{d\_high}], district_{id})$. Home owners define *mode goals* within their houses, to prioritise either "comfort" or "saving" modes - e.g., $G_{mode}(''comfort'', house_{id})$ and $G_{mode}(''saving'', house_{id})$. Mode goals can also be set on individual devices directly - e.g. $G_{mode}(''saving'', device_{id})$. Home owners can also specify power goals; for simplicity we only allow this for devices - $G_{power}([P_{h\_low}, P_{h\_high}], device_{id})$. Figure 4 shows how these goals are defined (in simplified notation) for one district and two houses.

Let us now identify the S_A and S_I for each one of these goals (steps 2-3). For simplicty we ignore context scopes. The district's power goal - $G_{power\_district}$ in Figure 4 - has an S_A that comprises all electric devices in the district. Yet, for legal reasons, it can only act directly at the house level; then each house acts on its own devices (as discussed later). For a house's mode goal - e.g. $G_{comfort\_house1}$ - the S_A covers all devices in the house. Each device's goal (power or mode) has an S_A that includes that device. Finally, all influence scopes S_I include all electric devices since these are all connected to the same district grid. In addition, the S_Is of mode goals include the atmosphere of targeted rooms and of neighbouring rooms.

For each goal, we now define the **abstract layers** and the corresponding goal-translation process (step 5). The controller pursuing the district power goal $G_{power\_district}$ has three abstract layers: district, house and device. Each layer is composed of one or several CEs: $CE_{district}$ in the district layer; CE_{house1} and CE_{house2} in the house layer; and $CE_{heater1}$, $CE_{heater2}$ and CE_{lamp} in the device layer [Fig. 4]. The CE in the district layer translates the district's power goal (declarative) into individual commands (procedural) for CEs in the house layer; and then into commands for CEs in the device layer. Commands are further discussed when defining the exact integration organisations. Device controllers pursuing mode goals comprise two abstract layers. Hence, each CE in the device layer is further split into these two abstract layers [Fig. 6]. For a heater, these translate mode goals into temperature intervals (predefined) and then into concrete power values (via a PID controller). For a lamp, mode goals are translated to light intensity then to power values.

Considering the architectural layout so far we decide to adopt a **hierarchical organisation** for **integrating CEs** within district, house and device layers (step 6). For the power goal, this organisation includes two roles: *power managers* that pursue power goals by orchestrating *prosumers*. In [Fig. 5] $CE_{district}$ plays a power manager role; device CEs prosumer roles; and CE_{house1} both roles - prosumer (for $CE_{district}$) and manager (for $CE_{heater1}$, $CE_{heater2}$ and CE_{lamp}).

We now define the power-related *commands* that are exchanged among roles in the organisation. When a manager detects that measures approach the power interval's high limit it sends a *reduce_load* order to its prosumers; for the lower limit it sends a *rise_load* order; when well in-between the limits it sends an *any_load* order to cancel the previous ones. To avoid oscillations, these orders are sent progressively, in random order, and the effects observed before new orders are sent. For mode goals, "comfort" or "saving" goals are simply transmitted from house CEs to device CEs.

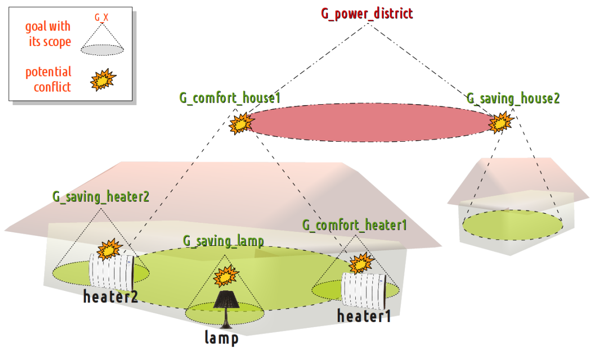


Figure 4: Goals and scopes in the smart micro-grid.

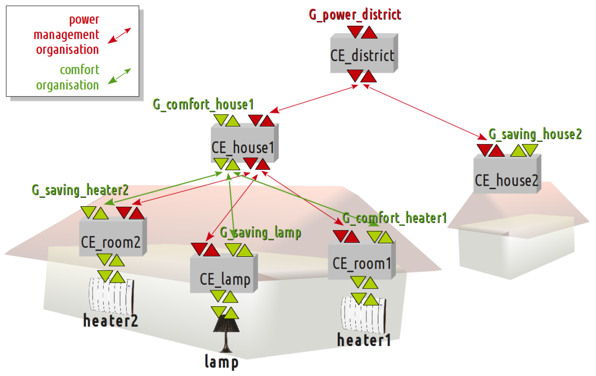


Figure 5: Integration organisations in the micro-grid.

To obtain a controller that pursues all presented goals simultaneously, the corresponding organisations are superposed onto one hierarchy. Here, CEs at each level combine the architectural layers of all previous organisations. Let's see some of the **conflicts** that can occur (step 7). One conflict is caused by district power goals and house mode goals intersecting over the house scope - the **conflict zone** includes all CEs and resources in the home, as they belong to the S_A of both goals. Since S_Is of all houses also intersect, house power goals are also conflicting. Yet, this conflict is equivalent to the previous one since house consumptions will be reflected in the district's power evaluation. Another conflict can occur between mode goals set for the entire house and directly on each device; here, the conflict zone includes the concerned device.

Both conflicts are **resolved** by adding **goal management layers** to CEs in the conflict zones [Fig. 6] (step 8). This control layer receives conflicting goals as inputs and provides a coherent goal as output for the control layer below - e.g. a power interval for power managers in house CEs; and, a temperature interval for PIDs in heater CEs. Goal managers give priority to mode goals. If in "comfort" mode, it ignores orders from power managers above; if "saving", it modifies the interval for the manager below depending on the orders received ([Fig. 7] and [Fig. 8]). In addition, goal managers of devices prioritise goals that are set on the device directly over those that are derived from the house mode goal.

Finally, the prototype does not include self-adaptation functions or further refinements (steps 9 and 10). From a design perspective, an adaptation-control layer could be added for instance to power management CEs, so as to discover and integrate new prosumers. Also, an adaptation layer could be added to the heater's PID abstract layer for reconfiguration purposes.

5.2 Scenarios, results and discussion

The scenarios depict the smart micro-grid when the outside temperature is dropping, heaters increase consumption thus rising the district load [11]. They focus on the behaviour of two district houses - h1 and h2. H1 is set to a "comfort" mode - most heaters ignore load-related orders and sustain a 23°C temperature; only a few that were directly set in "saving" mode respond. Hence, h1's power target is crossed [Fig. 8-a]. This conforms to the user's "comfort" goal and will impact the bill. The district manager detects a consumption increase and starts sending reduce_load orders to house prosumers. Since in "comfort" mode, h1's CE ignores them. H2 is initially set to "saving" and reacts by lowering its power interval [Fig. 8-b]; it then sends reduce_load orders to device prosumers to fit the new range. Heaters react by lowering their temperatures to 20°C which therefore lowers their consumptions and helps the district manager. To illustrate a **dynamic goal change**, let's assume that h2's

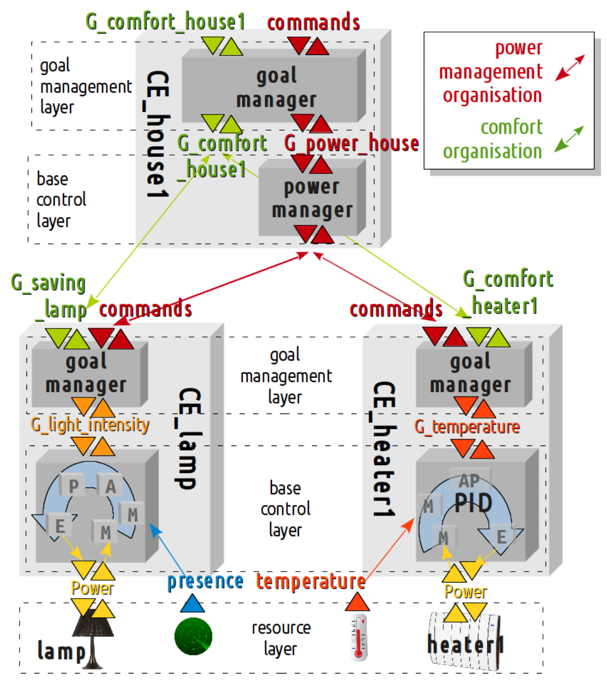


Figure 6: Design detail for the house control.

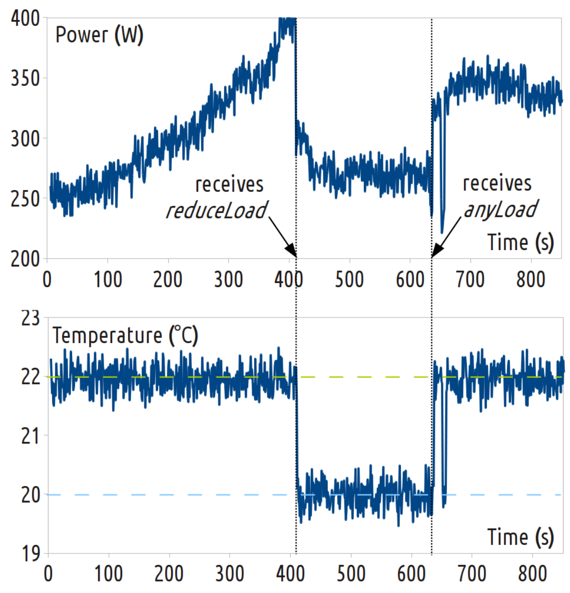


Figure 7: Management of a flexible heater.

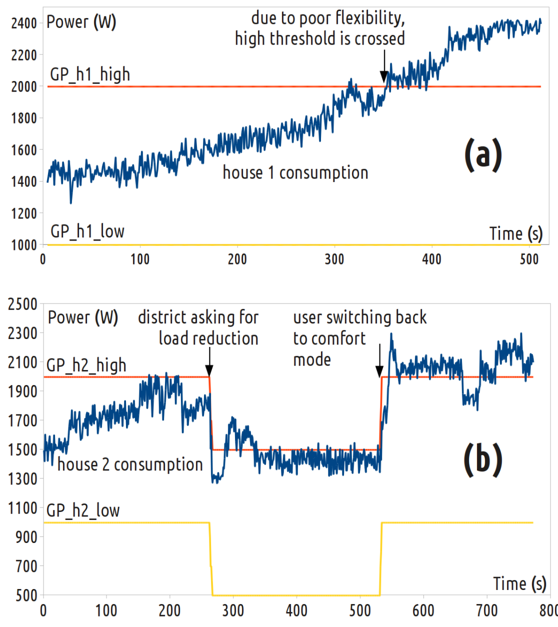


Figure 8: Power management in h1 (a) and h2 (b).

owners switch its mode to “comfort”, to accommodate an unexpected guest. H2 **self-adapts** - its prosumers ignore orders and consume more [Fig. 8-b].

The scenarios were run on a smart grid simulator that models physical entities, like houses, rooms, grid, heaters, lamps or solar panels; with related behaviours and attributes such as heat transfer, temperature and prosumption (in simulation time [s]). It is based on a service-oriented component technology - iPOJO/OSGi - and Akka middleware. These enable devices and CEs to be deployed, reconfigured and removed at run-time. A miniature house model was also built to ensure realistic behaviour. For limited space reasons, the presented results (based on the simulation) were selected for illustrative purposes; a web version of the simulation is available online for further explorations^2.

6 Related Work

This paper's contribution intersects several interrelated works, from various domains, from which we can only cite a few here. Separating goals from the means of achieving them has been proposed in autonomic computing [17], system engineering [21] and software agents [26]. In [17], management objectives can be defined as procedural policies, declarative goals or utility functions. In [21], “posed problems” are separated from the “resources” that can solve them, hence delaying resource selection until runtime. In the AI domain, “intelligent” agents modify the environment to achieve declarative goals [26]. They can adapt internal strategies when faced with unpredictable situations. In cybernetics, Ashby proposed an “ultra-stable” architecture that relies on two superposed control loops for adapting the system and its control strategy [2].

The generic architecture presented is consistent with these approaches - it generalises goal definitions to include scope and time (also identified in [18]), separates goals from control logic and introduces meta-control layers to self-adapt this logic and negotiate goals. Similar layered architectures have been proposed in various domains, including Brook's subsumption architecture in robotics [6] or Kramer and Magee's architecture [19] for autonomic systems. We drew inspiration from these proposals and identified the different natures of concerns that lead to system layering. The model thus proposes abstract, control and integration layers, which address orthogonal concerns and can be combined in recursive ways. The organisation paradigm is common in the Multi-Agent Systems domain [14][29] and was adopted in the model to enable internal adaptation and integration of plug and play resources while conserving important invariants. We are exploring this idea further in a parallel project [9]. In [10] we have presented an initial catalogue of organisations focused on conflict resolution. Splitting controllers into CEs of various types - such as control tasks and feedback loops - relies on previous projects [23]. The *feedback loop* appears as a first-class entity in all autonomic systems [7][15].

The generic architecture presented is complementary with many contributions that address particular issues of autonomic computing and IoT. These include numerous application-specific solutions that propose ad-hoc ways of constructing or integrating control-loops - e.g. monolithic control in DigiHome [25]; hierarchical managers in fANFARE [22], AutoHome [5], or using a single coordination manager [12]; or agent-oriented managers [16]. These fit the generic architecture, representing particular instantiations. Another category of complementary contributions focus on specific communication protocols and integration middleware for heterogeneous plug-and-play devices, like DigiHome [25] or RoSe [22] home automation platforms. Finally, [8] presents a generic integration model focused on categorising control loops based on their reciprocal interference (via shared knowledge) and proposing coordination and synchronisation protocols to integrate them.

Self-management requirements for the smart micro-grid have been identified in several works [3][4][13][27]. Notably, [3] proposes a distributed load management algorithm, where "colour" statuses solve management conflicts between load balancers and appliances. These fit the arhitectural model and can be adopted to implement corresponding CE layers. The smart grid domain was targeted here as a rich use case for illustrating the architecture and highlighting its main contributions.

7 Conclusions and Future Work

This paper proposed a generic architecture and methodology to help analyse and design multi-goal, multi-scale, adaptive autonomic control systems operating in distributed open environments, such as the IoT. It relies on the assumption that autonomic systems of this kind will be built by integrating control elements (CEs) of diverse types. Taking a SE-oriented approach, it aims to identify the reusable artefacts that can help instantiate this type of solution.

The contribution includes a conceptual model; a generic architecture adopting these concepts; and a development methology indicating how control applications can be designed guided by these concepts and architecture. The conceptual model considers *goals* as key elements that should be separated from the control logic necessary to pursue them. It provides a goal definition that can be translated, split and propagated across various CE types in the system, down to concrete actions on resources. The main difficulty factors are identified - including conceptual abstraction gaps, logistical complexity, adaptability and scalability issues - and suitable SE techniques identified for addressing them - including orthogonal types of layering and flexible modular architectures. The conceptual model also identifies *conflicts* as stand-alone elements that must be clearly defined, identified and addressed. The generic architecture relies on this conceptual base to define more concrete artefacts for system design. It includes several types of CEs - control tasks, integration tasks and control composites - featuring different behaviours and hence requiring different facilities - base, adaptive and agent-like functions. To integrate artefacts into flexible systems while ensuring core properties the model adopts an organisation-oriented approach inspired from the multi-agents. It indicates how this can be extended with reusable artefacts specific to conflict-resolution to handle multi-goal scenarios.

To illustrate its applicability and benefits the paper showed how a sample smart micro-grid was designed and implemented based on the generic architecture and methodology. Several runtime scenarios were selected to show how to define goals in business-specific terms, translate and split them among several abstraction levels, deal with multiple authorities and heterogeneous resources, handle multi-goal conflicts, adapt to dynamic context changes and goal reconfigurations, and integrate new resources. The paper did not address issues related to security concerns and the possible incompatibility of integrated CEs. System robustness and scalability were considered in the general architecture model but not yet tested or shown here.

Future work will concentrate on analysing autonomic systems in other domains to further test the architecture's applicability and extend it if necessary. This will include time-related concerns, which are critical to decision-making, decentralised coordination and system stability. The authors intent is to bring forward the understanding of autonomic systems operating in the IoT context and the associated support for developing them.

References

[1] Allerding, F., Becker, B., and Schmeck, H. Decentralised energy management for smart homes. In *Organic Computing A Paradigm Shift for Complex Systems*, C. Muller-Schloer, H. Schmeck, and T. Ungerer, Eds., vol. 1 of *Autonomic Systems*. Springer Basel, 2011, pp. 605–607.

[2] Ashby, W. R. *Design for a brain.* New York :Wiley, 1954.

[3] Beal, J., Berliner, J., and Hunter, K. Fast precise distributed control for energy demand management. In *Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2012, Lyon, France* (2012).

[4] Becker, B., Allerding, F., Reiner, U., Kahl, M., Richter, U., Pathmaperuma, D., Schmeck, H., and Leibfried, T. Decentralized energy-management to control smart-home architectures. In *ARCS* (2010), vol. 5974 of *Lecture Notes in Computer Science*, pp. 150–161.

[5] Bourcier, J., Diaconescu, A., Lalanda, P., and McCann, J. A. Autohome: An autonomic management framework for pervasive home applications. *TAAS 6*, 1 (2011), 8.

[6] Brooks, R. A. *Cambrian Intelligence: The Early History of the New AI.* MIT Press, 1999.

[7] Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., and Shaw, M. Software engineering for self-adaptive systems. Springer-Verlag, Berlin, Heidelberg, 2009, ch. Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70.

[8] de Oliveira Jr., F. A., Sharrock, R., and Ledoux, T. Synchronization of multiple autonomic control loops: Application to cloud computing. In *COORDINATION* (2012), M. Sirjani, Ed., vol. 7274 of *Lecture Notes in Computer Science*, Springer, pp. 29–43.

[9] Debbabi, B., Diaconescu, A., and Lalanda, P. Controlling self-organising software applications with archetypes. In *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on* (2012), pp. 69–78.

[10] Frey, S., Diaconescu, A., and Demeure, I. Architectural integration patterns for autonomic management systems. *9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe)* (2012).

[11] Frey, S., Huguet, F., Mivielle, C., Menga, D., Diaconescu, A., and Demeure, I. Scenarios for an autonomic micro smart grid. *1st International Conference on Smart Grids and Green IT Systems (SmartGreens 2012)* (2012).

[12] Gueye, S. M.-K., Rutten, É., and Tchana, A. Discrete control for the coordination of administration loops. In *IEEE Fifth International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA* (2012), pp. 353–358.

[13] Hermanns, H., and Wiechmann, H. Demand-response management for dependable power grids. In *Embedded Systems for Smart Appliances and Energy Management*, C. Grimm, P. Neumann, and S. Mahlknecht, Eds., vol. 3 of *Embedded Systems*. Springer New York, 2013, pp. 1–22.

[14] Horling, B., and Lesser, V. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev. 19* (December 2004), 281–316.

[15] IBM. An Architectural Blueprint for Autonomic Computing. 2006.

[16] Jennings, N. R., and Bussmann, S. Agent-based control systems. *IEEE Control Systems Magazine 23* (2003), 61–74.

[17] Kephart, J. O., and Walsh, W. E. An artificial intelligence perspective on autonomic computing policies. In *POLICY* (2004), IEEE Computer Society, pp. 3–12.

[18] Kernbach, S., Schmickl, T., and Timmis, J. Collective Adaptive Systems: Challenges Beyond Evolvability, 2009.

[19] Kramer, J., and Magee, J. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering* (Washington, DC, USA, 2007), FOSE '07, IEEE Computer Society, pp. 259–268.

[20] Lalanda, P., McCann, J., and Diaconescu, A. *Autonomic Computing: Principles, Design and Implementation*. Undergraduate Topics in Computer Science Series. Springer London, Limited, 2013.

[21] Landauer, C. Problem posing as a system engineering paradigm. In *ICSEng* (2011), H. Selvaraj and D. Zydek, Eds., IEEE, pp. 346–351.

[22] Maurel, Y., Chollet, S., Lestideau, V., Bardin, J., Lalanda, P., and Bottaro, A. fanfare: Autonomic framework for service-based pervasive environment. In *IEEE SCC* (2012), L. E. Moser, M. Parashar, and P. C. K. Hung, Eds., IEEE, pp. 65–72.

[23] Maurel, Y., Lalanda, P., and Diaconescu, A. Towards a service-oriented component model for autonomic management. In *IEEE SCC* (2011), H.-A. Jacobsen, Y. Wang, and P. Hung, Eds., IEEE, pp. 544–551.

[24] Ogata, K. *Modern Control Engineering*, 2nd ed. Prentice Hall PTR, 1990.

[25] Romero, D., Hermosillo, G., Taherkordi, A., Nzekwa, R., Rouvoy, R., and Eliassen, F. The digihome service-oriented platform. *Software: Practice and Experience* (2011).

[26] Russell, S. J., and Norvig, P. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education, 2010.

[27] Schmeck, H., and Karg, L. E-energy - paving the way for an internet of energy (auf dem weg zum internet der energie). *it - Information Technology 52*, 2 (2010), 55–57.

[28] Uckelmann, D., Isenberg, M.-A., Teucke, M., Halfar, H., and Scholz-Reiter, B. Autonomous control and the internet of things: Increasing robustness, scalability and agility in logistic networks. In *Unique Radio Innovation for the 21st Century*, D. C. Ranasinghe, Q. Z. Sheng, and S. Zeadally, Eds. Springer Berlin Heidelberg, 2011, pp. 163–181.

[29] Weyns, D., Haesevoets, R., Helleboogh, A., Holvoet, T., and Joosen, W. The macodo middleware for context-driven dynamic agent organizations. *TAAS 5*, 1 (2010).

Notes

[1]Software agent may be of these types [26], here we only use goal-oriented agents that can manage goals.

[2]try the simulation online at http://perso.telecom-paristech.fr/ sfrey/

Learning Deployment Trade-offs for Self-Optimization of Internet of Things Applications

Arun kishore Ramakrishnan, Nayyab Zia Naqvi, Zubair Wadood Bhatti,
Davy Preuveneers and Yolande Berbers
iMinds-DistriNet, Department of Computer Science
KU Leuven
3001 Leuven, Belgium

Abstract

The Internet of Things (IoT) is the next big wave in computing characterized by large scale open ended heterogeneous network of things, with varying sensing, actuating, computing and communication capabilities. Compared to the traditional field of autonomic computing, the IoT is characterized by an open ended and highly dynamic ecosystem with variable workload and resource availability. These characteristics make it difficult to implement self-awareness capabilities for IoT to manage and optimize itself. In this work, we introduce a methodology to explore and learn the trade-offs of different deployment configurations to autonomously optimize the QoS and other quality attributes of IoT applications. Our experiments demonstrate that our proposed methodology can automate the efficient deployment of IoT applications in the presence of multiple optimization objectives and variable operational circumstances.

1 Introduction

No doubt, recent advances in ICT have changed our verve enormously. Out of many emerging technologies there is a continuous rise of highly distributed ambient computing environments such as the Internet of Things (IoT) and the Machine-to-Machine (M2M) communication paradigm. IoT is an open ended network infrastructure with self-configuring capabilities fueled by low cost wireless communication and efficient network performance. It is a dynamic network of uniquely identifiable fixed or mobile communicating *objects*. These objects collect data, relay information to one another, process the information collaboratively, and take actions in an autonomic way without human intervention. Smart homes and offices, smart health, assisted living, smart cities and transportation are only a few examples of possible application scenarios where IoT is playing a vital role. Also in this domain many significant self-* challenges exist. For example, one challenge on self-optimization is how to change the behavior of a system to achieve a desired functionality, while maintaining a balance with Quality of Service (QoS) and resource usage [21]. Self-optimization in the Internet of Things shifts the focus from design and deployment of a single or a few elements operating autonomously to a large complex ecosystem of a network of autonomous elements [16].

Most of the existing software platforms for IoT are highly domain-specific prohibiting seamless interoperability of *objects* across multiple vertical domains. The FP7 BUTLER project[1] aims to address this concern by achieving a secure, context-aware horizontal architecture for IoT by offering common functionality on three platforms - *Smart Object*, *Smart Mobile* and *Smart Server*. In this work we aim to predict and control the global system behavior resulting from self-optimization of the components deployed among these three different platforms. The dynamic deployment of software components in an IoT system has to take into account the resource characteristics of the application components and the platforms used for deployment in terms of processing power, bandwidth, battery life and connectivity [1]. Each platform has its own capabilities and limitations to achieve Quality of Service (QoS) requirements. The heterogeneity makes it more complex and challenging to cope with QoS requirements.

The main objective of our work is to find optimal distributed deployments and configurations of application components. We use annotated component graphs to model application compositions and Pareto-curves to represent the optimization options for each (type of) platform, i.e. the *Smart Object*, *Smart Mobile* and *Smart Server*. The resource optimization objectives are chosen with respect to the QoS requirements and the trade-offs on the computation vs. communication cost-benefits. For the runtime (re)configuration and (re)deployment,

[1]http://www.iot-butler.eu/

we use Markov Decision Processes to achieve the self-optimization capabilities of the system.

After discussing related work in section 2, we present some motivating use cases in the healthcare and wellbeing domain in section 3 from which we elicit relevant functional and non-functional requirements. We briefly outline our self-optimization approach in section 4. It is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances. In section 5 we evaluate the deployment and optimization trade-offs in our work, and finally conclude this paper with possible directions for future work in section 6.

2 Related work

The autonomic computing paradigm has been around for almost a decade with a primary vision of computing systems that can manage themselves [10, 8]. This vision is now gaining inroads into the Internet of Things (IoT), with many typical optimization criteria:

- increase the performance by deploying heavyweight application components on faster hardware.
- reduce the amount of communication and network latencies between distributed components.
- optimize the overall energy consumption of the application components on the different platforms.

Utility functions are often used to achieve self-optimization in distributed autonomic computing systems, both for the initial deployment of an application and its dynamic reconfiguration. Tesauro et al. [19] explored utility functions as a way to enable a collection of autonomic elements to continually optimize the use of computational resources in a dynamic, heterogeneous environment. Later work by Deb et al. [5] investigated how utility functions can be used to achieve self-optimized deployment of computationally intensive scientific and engineering applications in highly dynamic and large-scale distributed computing environments. Utility functions have also found their way into the cloud computing space [7, 11] where they are used to manage virtualized computational and storage resources that can scale on demand.

The problem with utility functions is that their definitions require a fair amount of domain-specific knowledge to be effective. To address this challenge, reinforcement learning is often considered to automatically infer optimal deployment strategies. Tesauro [17, 18] explored reinforcement learning for an online resource allocation task in a distributed multi-application computing environment with independent time-varying load in each application. Similar work was proposed by Vengerov [20] using reinforcement learning in conjunction with fuzzy rulebases to achieve the desired objective. However, long training times is a reoccurring concern that often outweighs the potential benefits of reinforcement learning.

Organic Computing is another paradigm that focuses on distributed systems that exhibit self-* properties. In [3], a generic observer/controller architecture is proposed to introduce self-organization in complex systems such as traffic light controllers. The observer collects relevant data, pre-processes and analyzes it to discover patterns which might affect the performance of the system. The controller explores the parameter space to discover settings that would suit the future states of the system, but also matches the appropriate parameter settings to the current state of the system. For the traffic controller use-case, an evolutionary algorithm-based approach is used to explore and optimize the solution space and discover appropriate parameter settings. The controller then compares the performance of the discovered parameter settings in a simulation environment and deploys the most appropriate setting at runtime.

Similarly, in [15] the authors propose a new framework for self-organizing systems, albeit for improving the efficiency in terms of functional requirements of the system. In line with the observer/controller architecture proposed in [3], an advisor (a high-level agent) monitors the performance of other agents in a distributed environment and provides suggestions to improve their performance. The main focus of the paper is to improve the overall efficiency of the system considering the openness and autonomy of the system along with low observability and controllability of the agents (such as in the domain of pick-up and delivery). The advisor gathers data, analyzes and extracts recurring tasks and optimizes the solutions for those recurring tasks. In the aforementioned use case, exception rules are generated based on the current environmental conditions in order to improve the efficiency of the pick-up/delivery systems.

The focus of both the papers [3, 15] is on optimizing the functionality of the system while considering scalability and robustness requirements of the system. Contrary to our approach, the optimal system configuration for the architecture in [3] is completely determined online. Such an approach may require considerable resources at runtime and hamper the feasibility on resource constrained devices. Although [15] relaxes the need of continuous monitoring by providing some autonomy to the application for a limited amount of time, it does not address the performance/efficiency trade-offs which is of utmost importance in resource constrained IoT systems.

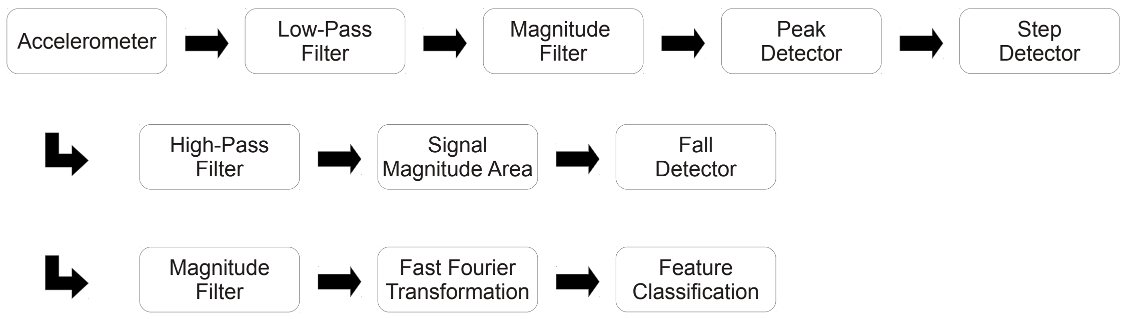


Figure 1: Component-based composition of the activity recognition application

Given the aforementioned optimization criteria, efficient deployment of application components in an IoT environment is often a multi-objective optimization problem [6, 13]. Note that these optimization objectives may conflict with one another (e.g. performance vs. energy consumption). In such cases, there does not exist a single solution that simultaneously optimizes each objective and resource trade-offs are to be made [9]. Pareto optimization [4, 22] is a technique that identities a set of Pareto-optimal solutions involving more than one objective function to be optimized simultaneously. We say that a solution − i.e. an allocation of resources − is Pareto-optimal if there exists no other alternative that would improve upon one objective function without deteriorating in at least one of the other objective functions.

On the one hand, the problem with utility functions (or optimization objectives) and Pareto-optimal solutions is that the Internet of Things is an open ended ecosystem of heterogeneous resources, making the crisp definition of Pareto-optimal solutions difficult due to an incomplete view on the external factors and uncertain circumstances that might influence the optimality. On the other hand, the applicability of the above learning approaches in an Internet of Things environment is usually hampered by the time and computational resources required to find a feasible or better solution. To address this concern, we aim to explore the feasibility of finding reasonable results in a reasonable amount of time by combining Pareto-optimization with reinforcement learning.

3 Scenarios and requirements for wellness and independent living

In this section, we will use some motivating scenarios from the healthcare and wellness domain as prototypical examples of IoT applications, and derive functional and non-functional requirements.

3.1 Use cases and components

Analysis of physical fitness and several health monitoring techniques revolve around the inference and prediction of human behavior. Accelerometer sensor data helps to analyze the human behavior in an effective way [14, 12]. We have implemented a variety of processing components in a modular fashion to enable a flexible deployment composition on the following platforms:

- *Smart Object*: Small appliances, sensors or actuators with limited computational power, storage capacity, communication capability, energy supply and primitive user interface are categorized as smart objects (e.g. RFID tagged objects, motion detectors, heating regulators).
- *Smart Mobile*: Devices with multi-modal user interfaces to enable user mobility through remote services are categorized as smart mobiles (e.g. smart phones, smart TVs). They usually have better resource provisions than smart objects.
- *Smart Server*: The aggregation and complex analysis of data from smart objects and smart mobiles are realized as services on smart servers (e.g. a local server or remote cloud computing set-up).

3.1.1 Use case 1 - motion activity recognition

In our first use case, we monitor the physical activity of the user by learning and classifying the activity of the user (e.g. standing, walking, running). We track the number of steps taken each day as a measure for wellbeing, and use it as input to classify higher levels of activity (e.g. cooking, watching TV, presenting at a meeting).

3.1.2 Use case 2 - fall detection for elderly

Another important parameter that characterizes the quality of independent life is the safety of the users in their own homes. Ageing can affect all domains of life leading to physical infirmity and loss of mental or cognitive abilities necessitating safety monitoring applications. Our second use case specifically focuses on fall detection as a common safety monitoring application within an Ambient Assisted Living (AAL) environment.

3.1.3 Application components

Both use cases leverage a tri-axial accelerometer, a common mobile embedded inertial sensor found in most smartphones, but rely on different sampling rates and processing algorithms. A conceptual overview of the software components is provided in Figure 1, with an explanation of some of them below.

- **Accelerometer**: It produces a continuous stream of X,Y,Z acceleration data by sampling the sensor at a certain rate (see Figure 2).
- **Low-pass filter**: For mobility tracking we are interested in acceleration peaks that arrive at a frequency of maximum 5Hz (i.e. max 5 steps per second). We use the 'moving average' as a simple low-pass filter to remove high-frequency noise (see Figure 2).

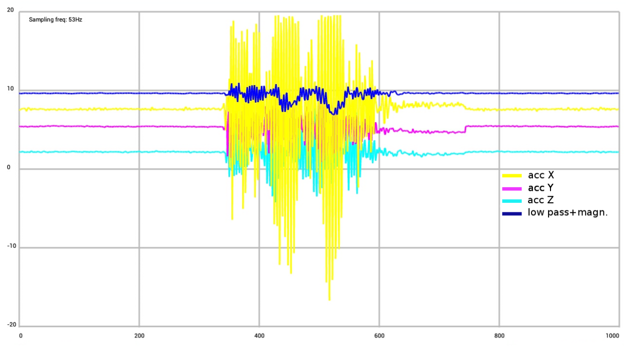


Figure 2: Accelerometer data and magnitude of signal after low-pass filter

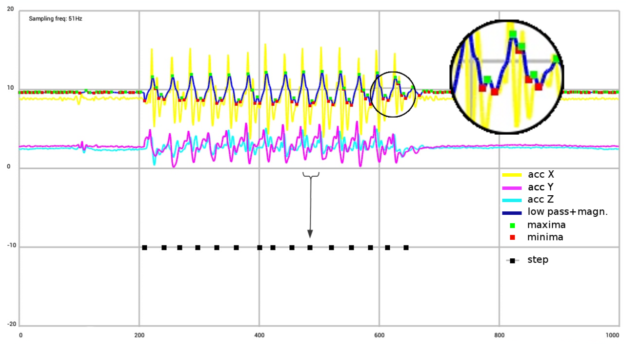


Figure 3: Peaks in magnitude signal and detected steps

- **Magnitude filter**: The orientation of the sensor is subject to change while moving around. Therefore, we carry out the signal analysis on the overall magnitude of the acceleration signal (see Figure 2).
- **Peak filter**: A single step is characterized by a pattern of several maxima and minima in the time domain of the acceleration signal. This component extracts these features in the signal for further analysis (see Figure 3).
- **Step detector**: It identifies the correct maxima/minima to correctly count the number of steps and to differentiate between standing still, walking and running (i.e. the peak rate) (see Figure 3).

Although this application is still fairly small in size and number of components, it manifests some interesting properties in the sense that the computational demands of certain components (e.g. the peak filter and step detector components) vary depending on the actual motion behavior of the user.

3.2 Requirements

The major (high-level) functional and non-functional requirements can be summarized as follows:

1. The system should be able to capture and store relevant sensor data and context information of the user to model, *learn, classify and predict the physical activity* of the users.
2. The system should have modular building blocks for data processing and activity recognition *on all three platforms for flexible distributed deployment*.
3. The deployment and configuration of the application components must *be adaptive at runtime to optimize for performance, latency, network communication (or QoS in general)*.

For example, delaying or offloading the accelerometer data processing will help to optimize the autonomy of battery powered sensors or mobiles.

Many opportunities for optimization may exist, i.e. different distributed deployments of the application components and different configurations per component. The challenge is to find and analyze the different optimization trade-offs in an open ended and dynamic IoT ecosystem of *Smart Objects*, *Smart Mobiles* and *Smart Servers*, each with varying sensing, communication, computation and storage capabilities.

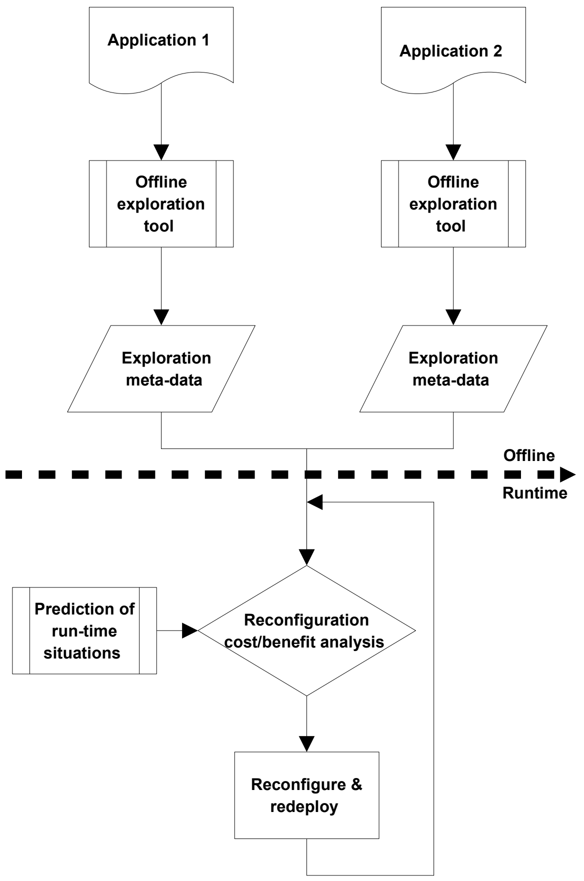


Figure 4: Overview of the self-optimization approach illustrating the offline and runtime phases

4 Conceptual overview of the deployment and optimization methodology

It is impossible to determine in advance where every component will run due to the dynamic interaction of these devices with the environment and the user. The multitude of parameters associated with the various possible configurations under varying workload and resource availability makes it almost impossible to manually finetune the components for best overall system performance, necessitating the introduction of self-* properties in IoT applications. Performing detailed cost benefit analysis for self-management decisions from scratch at runtime causes a large overhead. We reduce this overhead by balancing the offline and runtime efforts of making these decisions.

Our overall approach is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances. An overview of the approach is given in Figure 4.

4.1 Offline exploration of deployment and configuration options

Figure 5 gives an overview of the offline exploration for the preprocessing of deployment and configuration decisions. The component-based application is first profiled to obtain an annotated component graph. This annotated component graph is used for the exploration of the Pareto-optimal deployments and configurations and a reconfiguration cost matrix is constructed only for Pareto-optimal configurations. The runtime system uses the explored Pareto-optimal configurations and the reconfiguration matrices in order to make self-optimization decisions at runtime.

4.1.1 Deriving the annotated component graph

We use annotated component graphs as a high level model of computation to represent the application in order to explore the trade-offs between the different deployment configurations of the application. An annotated component graph is a directed graph where the nodes represent the components of an application, and the edges represent the data flow between the components. These nodes and edges are annotated with metadata representing the hard constraints, costs and resource requirements of the components.

Let us again consider the step counting application as an example. Some components of the application may be deployed on different platforms, i.e. a *Smart Object*, *Smart Mobile* and *Smart Server*. In order to generate an annotated component graph for this application the following steps are carried out:

1. Use the component model of the application and identity the data flows (similar to the one shown in Figure 1). The data flow graph acts as skeleton for the annotated component graph.

2. Instrument the communication interfaces of components to measure the amount of data transferred between components.

3. Run every component of the application on all the different platforms possible, profiling its execution time, energy consumption and data transferred between components, each time.

4. Calculate the memory requirements of every component by monitoring the changes in stack and heap sizes, as components are added and removed from the platform.

5. Repeat steps 3 and 4 over a range of component configurations (e.g. a different sampling rate) and/or simulated inputs (e.g. accelerometer traces of different activities and individuals).

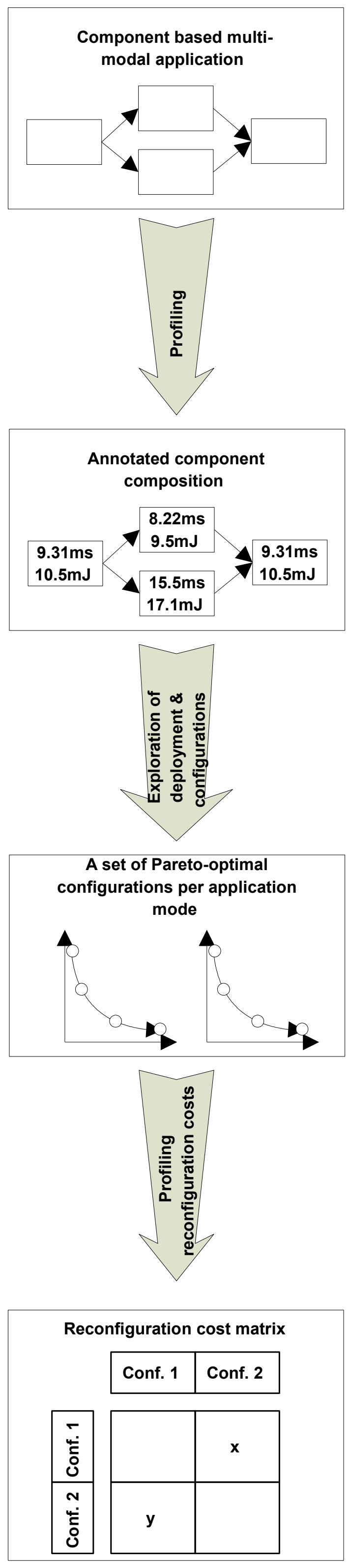

Component based multi-modal application
Profiling
Annotated component composition
8.22ms
9.5mJ
9.31ms
10.5mJ
15.5ms
17.1mJ
9.31ms
10.5mJ
Exploration of deployment & configurations
A set of Pareto-optimal configurations per application mode
Profiling reconfiguration costs
Reconfiguration cost matrix
Conf. 1
Conf. 2
Conf. 1
Conf. 2
x
y

Figure 5: Overview of the offline exploration phase

The only hard constraint for this application is that the accelerometer component can only execute on devices with such a sensor. Adding all this meta-data to the data flow graph generates the annotated component graph of the application and we use it as an intermediate model for exploring deployment trade-offs at design time.

Reconfigurable components: Some components have configuration options that affect their resource costs and requirements. For example, lowering the accelerometer sampling rate from 50Hz to 15Hz decreases the CPU time, communication and energy consumption of the activity recognition components, but increases the recognition error rate. For such components we annotate the component graph with metadata for a discretized range of parameter options, i.e. the CPU time and energy consumption values for the supported sampling rates.

Variability: Some components have stochastic non-functional performance properties (see Figure 6). For example, the communication throughput of a wireless node could be affected by external factors (e.g. interference). To define the Pareto-fronts (or Pareto-curves) one usually takes the worst case execution values after profiling to define the Pareto-points. Given that the IoT ecosystem is quite heterogeneous and open ended in nature, pursuing such a pessimistic approach will easily lead to undesirable solutions. Therefore, we define the Pareto-points based on the most likely execution values. However, to still be able to assess the impact of a worst case execution scenario for a particular deployment and configuration (i.e. a specific Pareto-point), we incorporate the likelihood distribution of the profiled execution values in each Pareto-point leading to a Pareto-front (i.e. a set of Pareto-optimal solutions) with some degree of variability.

4.1.2 Exploring the Pareto-optimal trade-offs

We model the problem of deploying an application to a heterogeneous network of self-managing *Smart Objects*, *Smart Mobiles* and *Smart Servers* as a constraint-based optimization problem and use a CPLEX based solver to explore the Pareto-optimal set of solutions. The details of expressing software deployment on hardware resources are described in our previous work [2].

In a Pareto-optimal set of solutions, every solution is better than all other solutions according to at least one functional or non-functional criterion. For example, Table 1 refers to a scenario of fancy and cheap hotels close to the beach. Hotels A, E and F can be eliminated because they are not Pareto-optimal. Also note that Hotel D is not the best in any optimization objective (stars, distance to beach and price), but it is Pareto-optimal. Although we are mainly interested in activity recognition

Hotel	Stars	Distance to beach	Price
A	**	0.7	80
B	*	0.2	40
C	***	1.3	100
D	**	0.3	70
E	**	0.5	90
F	**	1.5	120

Table 1: Maximization problem with multiple optimization criteria

as a motivating scenario, we use this example to offer a better understanding of Pareto-curves with multiple optimization criteria.

In our approach, eliminating deployment and configuration options that are not Pareto-optimal reduces the search space for the runtime reconfiguration decision from all possible configurations to the set of Pareto-optimal configurations. For example, consider the step counting application which consists of 5 components (see Figure 1) with a fairly simple pipe-and-filter architectural style. Assume we aim to deploy this application composition in a distributed setting on a simple sensor platform with limited processing capabilities (i.e. a *Smart Object*) and on a resource rich platform (i.e. a *Smart Server*). The deployment decision then boils down to figuring out which components are deployed on the sensor and which ones are deployed on the server. The only hard constraint for the deployment of this application is that the *Accelerometer* component must be deployed on the sensor platform. The other components can be deployed on either side, theoretically leading to 16 different deployment configurations of which a subset are Pareto-optimal. Obviously, extreme deployment configurations where components 1, 3 and 5 are on the sensor and components 2 and 4 are on the server will never be optimal due to the high communication cost.

In order to explore multi-dimensional Pareto-optimal surfaces, the problem is modeled using of parameterizable constraints. These parameters are then iteratively varied over a discretized range, invoking the solver each time to find a point on a Pareto surface. For example, an energy consumption versus Quality of Service Pareto curve is explored for the step counting algorithm by iteratively finding minimum energy solutions for different QoS constraints. It is important to note that there are no dependencies among the different invocations of the CPLEX solver. While finding solutions for this application takes several minutes on a single machine (depending on how many simulations are carried out), we can speed up this process by initiating parallel invocations of the CPLEX solver on a cluster of machines. This guarantees the feasibility of the approach for larger applications with many more configuration alternatives.

4.1.3 Reconfiguration cost matrix

A reconfiguration cost matrix is constructed by profiling the costs of reconfigurations and redeployments of components. For example, the cost of activation/deactivation of a component, establishing a local/remote component -to-component communication channel and transferring the state of an active component over a communication network. The size of this matrix is $O(N^2)$ where N is the number of possible configurations. As N can be become large, only the Pareto-optimal configurations are considered for reconfigurations.

4.2 Managing variability with runtime redeployment and reconfiguration

Traditionally, profiling of the application components is done with the assumption that each component will correspond to just one point in the Pareto search space. The openness in IoT can potentially create a lot of variability in the operational conditions of smart applications which in turn causes inconsistency in resource consumptions w.r.t. the Pareto-optimal solutions. For example, external environmental parameters such as network connectivity and communication bandwidth availability can vary depending on the living environment of the user. This operational variability makes it difficult to profile components in general. Similarly, the performance of an application component can vary depending on the user behavior. For example, the computational load of the *Step Detector* component (see Figure 1) will be different when the user is standing still (little processing due to no significant peaks in the accelerometer data) or walking (several peaks per second).

Rather than profiling application components as single points in the Pareto search space, we represent each application component with value distributions (through multiple profiling iterations) for systems where this variability in operational conditions is highly anticipated (as is the case for IoT systems). Each component is rep-

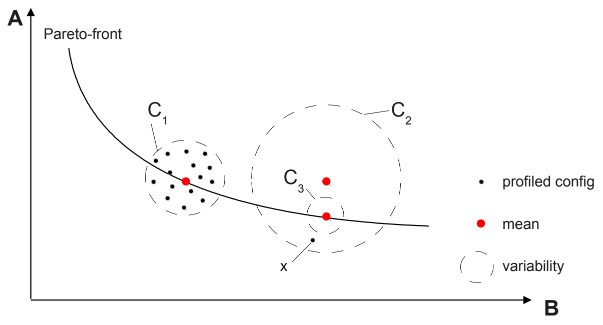


Figure 6: Variability in the profiled configurations

resented by statistical properties (e.g. min, max, median, average, standard deviation) discovered through multiple profiling iterations of the component under varying conditions. The example in Table 1 augmented with variable pricing (depending on season or room type) would require a similar Pareto-front representation.

We extended traditional Pareto-optimization methods to find a set of Pareto-optimal points taking into consideration this variability (see Figure 6). This basically means that the Pareto-optimal set not only considers configurations that are Pareto-optimal in the most likely scenarios (e.g. using the mean value of the optimization objectives), but also configurations that might be Pareto-optimal in less likely scenarios (e.g. min or max value of the optimization objectives). Assume in Figure 6 we aim to minimize for both objectives A and B. Configuration C_3 would be Pareto-optimal w.r.t. C_2 when considering the mean value of objective A (and the equal mean value for objective B). However, due to the difference in variability there might be a profiled configuration x of C_3 that has a lower value for objective A than any profiled configuration of C_2. Depending on which statistical property is chosen, we find different Pareto-optimal sets. In our approach, we take the union of these sets and refer to it as the *relaxed set of Pareto-optimal solutions* for any given statistical property. At runtime, we start off with a default statistical property to define the Pareto-optimal solutions, and propose using online reinforcement learning to discover whether the given context gives rise to other Pareto-optimal solutions that emerge in less likely situations. The major benefits of our approach are:

- Reduction of the (re)configuration search space by limiting the relevant working configurations to Pareto-optimal solutions.
- A modified Pareto-optimization method defining a relaxed set of Pareto-optimal solutions to handle the variability in the IoT working conditions.
- Finetuning the configuration at runtime by narrowing down the operational variability through reinforcement learning.

4.3 Analyzing cost/benefit trade-offs with Markov Decision Processes

With the relaxed set of Pareto-optimal solutions, we can find configurations that are optimal in a particular context. Whether these configurations remain beneficial over a certain time period is something we cannot infer from the Pareto-fronts.

Let us consider the 5 components in the step counting application in Figure 1 and the different deployment configurations. Whether any of these configurations remains optimal over time is unpredictable, and cannot be derived just from the offline generated Pareto-fronts. For example, the default sampling frequency for counting steps is set to 50Hz. However, if the system knows the person is not moving (e.g. sitting down in a meeting), it can reduce resource consumption by changing the configuration of the *Accelerometer* component and setting the sampling frequency to 15Hz. In this mode, it can detect a change in movement, and if so, set the sampling frequency back to 50Hz to start counting steps again.

We therefore model the relaxed Pareto-optimal configurations as states of a Markov Decision Process (MDP) along with the associated set of actions and rewards and find out the best possible (re)configuration policy over a finite time period. This uncertainty in potential benefits over time is introduced by a changing context in the operating environment of the system. Also note that these reconfigurations have associated reconfiguration costs which in turn would require the system to maintain the new configuration for a certain time T_{be} (break-even time) before it is actually able to benefit from deploying the new configuration.

As typical user activities are characterized by certain events that happen over certain period of time, the states are not expected to change at each time step. State transitions will be guided by transition rates, i.e. how quickly a transition takes place instead of how likely transitions are at each time step. Accordingly, a continuous time Markov process is ideal to model this problem but in order to reduce the complexity of the proposed system we have decided to utilize a discrete finite horizon MDP instead of a continuous MDP.

A classic discrete MDP is represented by a 4-tuple $S, A, P(s,s'), R(s,s')$ where S is the set of states, A is a super-set of sets of actions possible in each state, $P(s,s')$ is the transition probability between states s and s', and $R(s,s')$ is the reward for moving from state s to s'. The goal here is to discover and learn the expected rewards and best possible policy considering the transitions between configurations due to a changing context. The different parameters of our proposed MDP model are:

- **States**: a set of Pareto-optimal configurations for each application that can be possibly deployed in the system. It is represented as an n-tuple where n represents the number of platforms. If there are m possible configurations for each of the platforms, then the number of possible states is (m^n). Also, note that the momentarily Pareto-optimal global configurations are a small subset of these states.

- **Actions**: a set of possible state transitions that are allowed for optimal resource consumption are modeled as actions for each state, i.e. $a(s)$. We assume that all the application components have to be run on one of the available platforms.

- **Transition probability**: the probability with which state changes are anticipated in the system is called the transition probability. User activities or changing living conditions introduce randomness and cause deviations in the desired state transitions for the most likely Pareto-optimal solution.

- **Reward function**: the reward value is defined in terms of the resource consumption of the current and optimal configuration, and the time the optimal configuration will be active:

  $$\frac{resource\ consumption_{opt.\ conf.}}{resource\ consumption_{curr.\ conf.}} * T$$

  The resource consumption of a configuration on a particular platform is a weighted average of the m resources ρ_i involved:

  $$resource\ consumption = \sum w_i * \rho_i \qquad 0 < i < m$$

  We also assume that $T > T_{be}$, meaning that the cost incurred by the reconfiguration to the optimal one will be accounted for by running this new optimal configuration longer than the break even time T_{be}.

  The optimal configuration is the one which maximizes the value η defined as follows:

  $$\eta = \frac{\delta}{\sum w_{i,j} * \rho_{i,j}} \qquad 0 < i < m, 0 < j < n$$

  where,

  δ = accuracy of the step detector
  $\rho_{i,j}$ = resources consumed (in %)
  i = type of resource (memory, CPU, etc.)
  j = type of platform (smart object, mobile or server)
  w = weight to prioritize importance of resource

The above parameters can be explored offline to identify an optimal configuration for a given set of resources whose importance can be balanced by the user (e.g. battery and performance). However, the variability in the execution makes it impossible to guarantee that the aforementioned optimal solution will remain the same under all circumstances. We therefore use learning techniques to better identify the optimal deployment for the given operational circumstances of the application. The learning we propose is guided by an ε-greedy algorithm:

$$Q(s_{t+1}, a_{t+1}) = \varepsilon.mean(Q(s,a)) + (1-\varepsilon).max(Q(s,a))$$

where the first term helps the system to explore the relaxed Pareto-optimal configuration space and the second term exploits the learned best policy available at the time.

4.4 Discussion

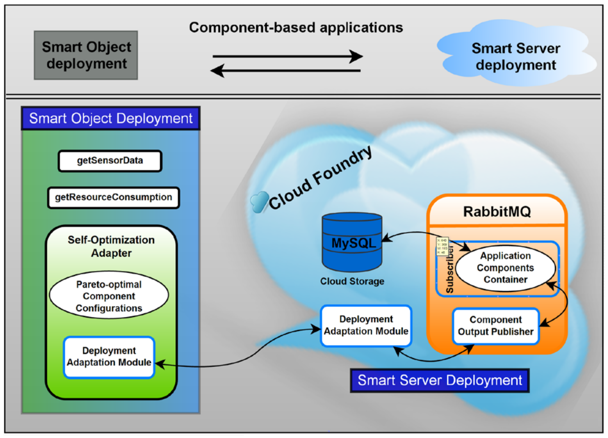
Despite pre-optimizing the deployment decision of the IoT application components and implementing application specific optimization logic, the global optimal configuration cannot be determined during the offline exploration phase as it is dependent on multiple time-varying variables such as, user profile (e.g., age and other factors can influence how active a person is) preference of the user (e.g., to minimize computational load or communication bandwidth) and the operating environment (e.g., signal strength of the WiFi network). Hence in this paper, a smart adapter meta-component is proposed which implements a global runtime learner to drive the IoT application towards optimal configuration over time for any given user or operating conditions. The local application specific optimization logic (e.g., lowering the sampling frequency when the user is not active) takes the current (or aggregated) prediction of the application (e.g., the user is idle or active) as input and output the optimal configuration for the system. A simple version can be implemented by a look-up table with pre-defined output events and corresponding optimal configurations. Whereas the smart adapter is more generic, it takes current configuration of the IoT application and corresponding resource consumption in multiple platforms as input and recomputes the throughput of the predefined configurations which in turn is used by the reinforcement learning algorithm to learn the best policy for the user and associated operating environment. Given that the learned optimal configuration policy is tailored for the user, it will overrule the policies determined by the offline exploration phase. As the resource needs of the smart adapter is pre-determinable (due to its fixed introspection frequency), it is modeled by the reconfiguration matrix and the associated cost is considered in the overall efficiency of the application.

5 Experimental evaluation

We will demonstrate the feasibility of our approach with use case 1 (the step counting application as shown in Figure 1), and evaluate the proposed methodologies using the 5 components. This simple deployment scenario allows different deployment compositions on three different platforms *Smart Object*, *Smart Mobile* or *Smart Server*. For the sake of simplicity, we will only use two platforms in our experiments (see Figure 7):

Smart Object: We use a SunSPOT development board[2] with a 400MHz ARM 926ej-S processor with 1MB RAM and 8MB flash memories. The processor runs applications on top of a Java "Squawk" virtual machine.

[2]http://www.sunspotworld.com/docs/Yellow/eSPOT8ds.pdf



Smart Object deployment
Component-based applications
Smart Server deployment
Smart Object Deployment
getSensorData
getResourceConsumption
Self-Optimization Adapter
Pareto-optimal Component Configurations
Deployment Adaptation Module
Cloud Foundry
MySQL
Cloud Storage
RabbitMQ
Subscriber
Application Components Container
Deployment Adaptation Module
Component Output Publisher
Smart Server Deployment

Figure 7: Experimental setup

Component	CPU load	Communication
Accelerometer	8.09 ± 1.3 ms	5.5 ± 0.0 kB/sec
Low-pass filter	57.9 ± 2.1 ms	5.5 ± 0.0 kB/sec
Magnitude filter	18.2 ± 1.5 ms	1.8 ± 0.0 kB/sec
Peak detector	14.9 ± 9.7 ms	0.5 ± 0.4 kB/sec
Step detector	5.12 ± 4.8 ms	0.1 ± 0.1 kB/sec

Table 2: Performance benchmark of the individual components on the sensor

The board has an integrated IEEE 802.15.4 compliant Radio Transceiver CC2420 from Texas Instruments.

Smart Server: Our Smart Server infrastructure runs VMware's open source Platform-as-a-Service (PaaS) offering known as Cloud Foundry on a server with 8GB of memory and an Intel i5-2400 3.1GHz running a 64-bit edition of Ubuntu Linux 12.04. Cloud Foundry provides messaging and database servers as built-in services. We deployed its open source distribution, i.e. VCAP[3]. VCAP supports the AMQP-based *RabbitMQ*[4] server for messaging and *MySQL* for storage and persistence. All of the configuration is done in *Spring*, an application development framework. Finally, we exposed our loosely coupled application components as services, integrating *Apache CXF* with the Spring framework.

We profile the step counting components under different deployment and configuration scenarios with an objective to optimize the CPU load and the network communication costs. The results of the profiling on the sensor are shown in Table 2. Note that for the *Accelerometer, Low-pass filter* and *Magnitude filter* components there is little to no communication variability because the amount of data output is fixed and depending on the sampling rate of the accelerometer.

[3]https://github.com/cloudfoundry/vcap
[4]http://www.rabbitmq.com

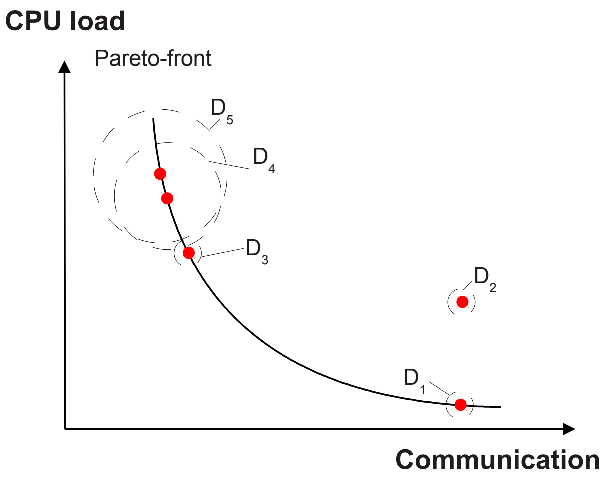


Figure 8: CPU load and network communication deployment trade-offs on the sensor

Given the fact that the deployment of the *Accelerometer* component is fixed, we have 16 different deployment options for the 4 remaining components. Some of the Pareto-optimal deployment options are (see Figure 8):

- D_1: Minimal computation on the sensor by having the *Accelerometer* component on the sensor and the 4 remaining sensor data processing components deployed on the server.
- D_2: The *Accelerometer* and *Low-pass filter* components deployed on the sensor and the other components on the server.
- D_3: The *Accelerometer*, *Low-pass filter* and *Magnitude filter* components deployed on the sensor and the other components on the server.
- D_4: All components except the *Step Detector* component deployed on the sensor.
- D_5: Highest CPU load on the sensor by having all the components deployed on the sensor and no communication cost between the sensor and the server.

Note that each deployment D_x represents the joint resource consumption and variability of the components deployed on the sensor. Examples of non-Pareto-optimal solutions include a.o. a deployment with the *Low-Pass Filter* and *Peak Detector* components on the server and the *Accelerometer*, *Magnitude Filter* and *Step Detector* components on the sensor. This mixed deployment causes a high communication cost.

We have also Pareto-fronts specifically for component reconfigurations. For example, the *Accelerometer* component can sample data at different rates, causing different CPU loads and communication throughput. Figure 8 shows the results of sampling at 50Hz, whereas a profiling at 15Hz produces a similar deployment trade-off but with an overall lower CPU load and network communication. The fall detection use case requires a 100Hz sampling rate, but involves different components with corresponding Pareto-fronts.

5.1 Resource-driven deployment trade-offs

In a first experiment, we tested the automatic deployment of our application components with an initial deployment D_1. The optimization policy was set to reduce the energy consumption, which automatically triggered the deployment of all the components except the *Step Detector* component on the sensor (configuration D_4). We then changed the optimization policy to minimize network communication (cfr. a GPRS communication scenario that incurs a real financial cost). At this point, the deployment of the latter component was also moved to the sensor (configuration D_5).

5.2 Contextual configuration trade-offs

In a second experiment with periods without motion activity, the system learned that the stationary state of the individual would last for at least 10 minutes. In this state, there were no more peaks detected leaving the *Step Detector* component idle. This exceptional circumstance (i.e. no communication to this component) triggered the component to be deployed again on the server (configuration D_4), lowering the sampling frequency to 15Hz, and switching to the corresponding Pareto-front.

5.3 Learning self-optimization trade-offs

The effect of the *Peak Detector* in the Pareto-search space is more fuzzy compared to the three previous components in the processing chain (whose CPU load and communication variability is low). The variability in the resource consumption of the *Peak Detector* component is due to external factors. For example, for elderly people the number of peaks would be smaller as they are less mobile. For more active young people, there are much more peaks to process. Hence, it is not clear-cut anymore to decide where to run this component as the decision is tied to individual users and their life-style. Furthermore, this contextual dependency cannot be captured in the Pareto-fronts through profiling. In a third experiment, we tested the self-optimizing capabilities of the MDP on an individual with a sedentary lifestyle. The MDP picked up this behavior after on average 110 iterations, and finetuned the Pareto-curve with lower computation and network communication variability for deployment solutions D_4 and D_5, leading to an overall preference for the latter deployment.

6 Conclusions

In this paper, we presented our self-optimization approach for deploying IoT application components. The goal is to autonomously find the trade-offs between different component deployment configurations and their resource impact for distributed deployments on *Smart Objects*, *Smart Mobiles* and *Smart Servers*. Our approach is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances.

Our experiments have shown that the deployment and configuration decision (which part of the application is run on a sensor, mobile or a server in the cloud) is not always clear-cut, and that trade-offs are to be made w.r.t. application and QoS requirements. Our modular design philosophy for developing IoT applications helps to dynamically configure, compose and deploy these components depending on the QoS requirements of the applications. We have profiled and benchmarked these components on different deployment ends. This helped us to automatically find trade-offs for a distributed deployment of these components considering both the performance impact as well as the cost/benefit of any reconfiguration or change in component deployment.

As future work, we will explore the effects of more advanced learning and classification techniques and broaden our methodology to validate more complex deployment scenarios.

References

[1] Bandyopadhyay, D., and Sen, J. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications 58* (2011), 49–69.

[2] Bhatti, Z., Miniskar, N., Preuveneers, D., Wuyts, R., Berbers, Y., and Catthoor, F. Memory and communication driven spatio-temporal scheduling on mpsocs. In *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on* (30 2012-Sept. 2), pp. 1–6.

[3] Branke, J., Mnif, M., Muller-Schloer, C., and Prothmann, H. Organic computing-addressing complexity by controlled self-organization. In *Leveraging Applications of Formal Methods, Verification and Validation,*

2006. ISoLA 2006. Second International Symposium on (2006), pp. 185–191.

[4] CENSOR, Y. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization 4* (1977), 41–59.

[5] DEB, D., FUAD, M. M., AND OUDSHOORN, M. J. Achieving self-managed deployment in a distributed environment. *J. Comp. Methods in Sci. and Eng. 11*, 3, Supplement 1 (Aug. 2011), 115–125.

[6] DEB, K. Multi-objective optimization. In *Search Methodologies*, E. K. Burke and G. Kendall, Eds. Springer US, 2005, pp. 273–316.

[7] HU, Y., WONG, J., ISZLAI, G., AND LITOIU, M. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research* (Riverton, NJ, USA, 2009), CASCON '09, IBM Corp., pp. 101–111.

[8] HUEBSCHER, M. C., AND MCCANN, J. A. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 7:1–7:28.

[9] KEENEY, R. L., AND RAIFFA, H. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993.

[10] KOEHLER, J., KOEHLER, J., GIBLIN, C., GIBLIN, C., GANTENBEIN, D., GANTENBEIN, D., HAUSER, R., AND HAUSER, R. On autonomic computing architectures.

[11] KOEHLER, M., AND BENKNER, S. Design of an adaptive framework for utility-based optimization of scientific applications in the cloud. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing* (Washington, DC, USA, 2012), UCC '12, IEEE Computer Society, pp. 303–308.

[12] KWAPISZ, J. R., WEISS, G. M., AND MOORE, S. A. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl. 12*, 2 (Mar. 2011), 74–82.

[13] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26* (2004), 369–395.

[14] RAVI, N., DANDEKAR, N., MYSORE, P., AND LITTMAN, M. L. Activity recognition from accelerometer data. In *Proceedings of the 17th conference on Innovative applications of artificial intelligence - Volume 3* (2005), IAAI'05, AAAI Press, pp. 1541–1546.

[15] STEGHOFER, J.-P., DENZINGER, J., KASINGER, H., AND BAUER, B. Improving the efficiency of self-organizing emergent systems by an advisor. In *Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on* (2010), pp. 63–72.

[16] SUNDMAEKER, H., GUILLEMIN, P., FRIESS, P., AND WOELFFLÉ, S. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things, European Commision* (2010).

[17] TESAURO, G. Online resource allocation using decompositional reinforcement learning. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 2* (2005), AAAI'05, AAAI Press, pp. 886–891.

[18] TESAURO, G. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing 11*, 1 (Jan. 2007), 22–30.

[19] TESAURO, G., AND KEPHART, J. O. Utility functions in autonomic systems. In *Proceedings of the First International Conference on Autonomic Computing* (Washington, DC, USA, 2004), ICAC '04, IEEE Computer Society, pp. 70–77.

[20] VENGEROV, D. A reinforcement learning approach to dynamic resource allocation. *Eng. Appl. Artif. Intell. 20*, 3 (Apr. 2007), 383–390.

[21] VERMESAN, O., FRIESS, P., GUILLEMIN, P., GUSMEROLI, S., SUNDMAEKER, H., BASSI, A., JUBERT, I., MAZURA, M., HARRISON, M., EISENHAUER10, M., ET AL. Internet of things strategic research roadmap. *Internet of Things: Global Technological and Societal Trends* (2009), 9.

[22] ZITZLER, E., LAUMANNS, M., AND THIELE, L. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems* (Athens, Greece, 2001), K. C. Giannakoglou, D. T. Tsahalis, J. Périaux, K. D. Papailiou, and T. Fogarty, Eds., International Center for Numerical Methods in Engineering, pp. 95–100.

Autonomic Fail-over for a Software-Defined Container Computer Network

Chien-Yung Lee[+], Yu-Wei Lee[+], Cheng-Chun Tu[*+], Pai-Wei Wang[+], Yu-Cheng Wang[+], Chih-Yu Lin[+], and Tzi-cker Chiueh[*+]

[*]Computer Science Department, Stony Brook University
[+]Cloud Computing Center for Mobile Applications, Industrial Technology Research Institute

Abstract

The ITRI container computer is a modular computer designed to be a building block for constructing cloud-scale data centers. Rather than using a traditional enterprise data center network architecture, which is typically based on a combination of Layer 2 switches and Layer 3 routers, the ITRI container computer's internal interconnection fabric, called *Peregrine*, is a software-defined network specially architected to meet the scalability, fast fail-over and multi-tenancy requirements of these data centers. *Peregrine* uses as the underlying physical interconnect a mesh of commodity off-the-shelf Ethernet switches, and adopts a centralized network control architecture that operates these Ethernet switches as a coordinated distributed data plane. Compared with vanilla enterprise networks, *Peregrine* features a fast fail-over capability not only for network switch/link failures, but also for failures of its own control servers. This paper describes the design and implementation of *Peregrine*'s fault tolerance mechanisms, and shows their effectiveness using empirical performance measurements taken from a fully working *Peregrine* prototype under various failure scenarios.

1 Introduction

The ITRI container computer is designed to be a modular building block for constructing a *cloud data center computer*, which, in the most general form, is composed of multiple container computers that are connected by a data center network, is interfaced with the public Internet through one or multiple IP routers, and is designed as an integrated system whose hardware components such as servers and switches are stripped off unnecessary functionalities, whose resources are centrally configured, monitored and managed, and which encourages system-wide optimizations to make the best end-to-end tradeoffs. One key design decision of the ITRI container computer is using only commodity hardware, including compute servers, network switches, and storage servers, and leaving high availability and performance optimization to the system's software. Another key decision is to design a new data center network architecture from the grounds up to meet the unique requirements imposed by a cloud data center computer. We named this data center network architecture *Peregrine* [5]. This paper focuses on the design, implementation and evaluation of the fault tolerance mechanisms in *Peregrine*.

Although *Peregrine* uses commodity off-the-shelf Ethernet switches as basic building blocks, it follows a software-defined network (SDN) [4, 8] design philosophy by doing away with most of the control plane functionalities in these switches and using a centralized network control server to operate these switches, and eventually turning them into a coordinated distributed data plane. *Peregrine* chooses this centralized control plane architecture because it offers two important advantages. First, it enables *Peregrine* to make more efficient use of all physical links in the underlying network. Second, it significantly reduces the fail-over latency associated with any single network switch/link failure.

Despite various optimizations, standard Ethernet-based networks take at least a few seconds to recover from a network switch/link failure, especially for large networks, because their normal operation assumes a spanning tree overlaid on top of the physical network, and re-building this spanning tree after a failure takes time. A fail-over latency of several seconds is not acceptable in large-scale data centers that are built out of commodity hardware components, because in these data centers HW failures are not uncommon and they need to be effectively masked so as to be completely hidden from applications and their users.

The fail-over latency goal of the ITRI container computer is set to 100 ms, which is set so as to mask each network failure event as a transient congestion. To achieve this goal, *Peregrine* does away with the concept of spanning tree completely, and therefore does not need to rebuild anything after a failure; moreover, it pre-computes and pre-installs a contingent plan for all nodes that are affected by every possible network switch/link failure to route around the failure, thus greatly reduces the fail-over latency to the minimum. This fast fail-over strategy is made possible by the centralized control plane architecture, because it is equipped with a global knowledge of the physical network topology, its up-to-date health status, and the network flows provisioned on them.

However, a centralized control plane is in theory more brittle because it is a single point of failure, that is, any control plane failure could potentially bring down the entire network. To overcome this issue, *Peregrine*'s control servers are designed to be fully redundant and thus highly available. The interaction of the high availability (HA) mechanism of *Peregrine*'s control servers with *Peregrine*'s fail-over mechanism is complex and subtle, and requires careful considerations to every low-level detail.

Finally, because the servers used in the ITRI container computer are also of commodity grade, the failure rate of these servers is not negligibly low. To enable seamless fail-over of application VMs running on these servers, *Peregrine* informs clients that interact with failed VMs and redirects them to their backup VMs that take over.

2 Related Work

There has been extensive studies on the resiliency of conventional Internet [10, 13]. These works compute a number of node or link disjoint paths between pairs of end points and switch over to its corresponding backup path upon link or switch failures. Provider Backbone Bridge Traffic Engineering (PBB-TE) swaps the B-VID value to redirect the traffic onto the pre-configured path within 50 ms under a path failure [3] . MPLS-TE [11,17] offers fast reroute functionality by redirecting encapsulated traffic to a backup path when the primary one fails. Mechanisms for monitoring and discriminating against intermittent link failures to achieve network stability are also addressed in [1, 16].

Numerous data center network architectures propose the fault tolerant data plane by introducing a centralized controller. PortLand [15] employs a centralized fabric manager and relies on switches to detect and inform its centralized fabric manager when a link or switch fails. The fabric manager maintains a fault matrix with per-link connectivity and informs affected switches to reroute packets. VL2 [9] depends on OSPF to re-converge quickly and allows applications to fully use a link several seconds after it is restored, due to the conservative defaults for OSPF timers. VL2's directory server also incorporates the asynchronous replicated state machine to offer a strongly consistence based on the Paxos consensus algorithm.

SDN proposes separation and centralization of the control plane from the data plane. Most of the existing OpenFlow-based SDN proposals address resiliency at either the controller [12, 20] or the data plane. Onix [12], a distributed control platform, provides coordination facilities for detecting and reacting to Onix instance failures. FlowVisor [18] partitions the underlying network and allows multiple controllers to manage their own slice of network. The data plane reliability relies on either the controller *proactively* pre-computes and pre-installs the backup paths on an OpenFlow switch [14] or *reactively* takes action upon receiving failure notifications [19]. For example, NOX [20] depends on existing switch mechanisms to determine link failures, notify NOX, and flushes the flow entries at that switch which use the failed link. However, the proactive mechanism requires installing additional flow entries into the OpenFlow switches, which has very limited TCAM entries, whereas the reactive mechanism incurs high latency. Moreover, one of the major concerns about SDN's split architecture design is the resiliency between the centralized controller and switches [19]. That is, any failure that disconnects the data plane form the control plane may bring down the entire network [2, 21]. Existing SDN proposals depends on an out-of-band control network to guarantee reachability between switches and the controller. However, the fail-over latency between controller and switches is usually at the timescale of seconds, due to the fact that the control network is running conventional distributed protocols such as spanning tree protocol (STP), IS-IS, or OSPF.

We argue that a well-architected SDN should have its fast fail-over mechanism among the data plane, the controller, and the control plane. *Peregrine* takes the first step in addressing all these three aspects using standard Ethernet switches and in-band control design.

3 Fault Tolerance Support in Peregrine

3.1 ITRI Container Computer

The ITRI container computer is physically housed in an ISO-standard 20-foot (6.096 meter) shipping container, and consists of 12 server racks lined up on both sides of the container with an access aisle in the middle, where each server rack holds up to 96 current-generation X86 CPUs and 3TB of DRAM. Twelve JBOD (Just a Bunch Of Disks) storage servers, each packed with 40 disks, are installed in the container computer. Together with the local disks directly attached to compute server nodes, the container boasts of more than 1 petabyte worth of usable disk space.

The ITRI container computer's network is a modified Clos network. Every rack contains 48 server nodes, each having 4 1GE NICs, and includes 4 top-of-rack (*TOR*) switches, each having 48 1GE ports and 4 10GE ports. There is a virtual switch inside every server node that is connected to the server node's four NICs, which in turn are connected to the four TOR switches in the same rack. The four 10GE unlinks on each TOR switch are connected to four different *regional* switches, each of which has 48 10GE ports. To improve the performance of storage accesses, each storage server has four 10GE NICs and is directly connected to four different regional switches. In total, five regional switches per rack are used in the ITRI container computer.

Peregrine [5] is the internal network for the ITRI container computer, and is built on commercially off-the-shelf Ethernet switches with most of their built-in control plane functionalities such as spanning tree protocol, source learning, flooding if unknown destinations, etc., turned off. Instead, *Peregrine* uses a centralized control plan that manages the forwarding tables of the underlying Ethernet switches. The software architecture of *Peregrine* is shown in Figure 1, and consists of a *kernel agent* that performs ARP query packet interception and transformation and is installed in the Dom0 VM of every Xen-based physical server, a *centralized directory server* (DS) that performs generalized IP to MAC address look-up, and a *centralized route algorithm server* (RAS) that constantly collects the network's traffic matrix, runs a load-based routing algorithm based on the traffic matrix, and populates the switches' forwarding tables with the computed routes. After the RAS computes routes for physical server pairs, it builds up an *inverse* map that associates every network link with all the computed routes that go through the link.

All packets from a DomU VM pass through the *Peregrine* agent in Dom0 of the corresponding physical machine. For each packet going by, the *Peregrine* agent consults with its local ARP cache with the packet's destination IP address, submits a lookup request to the DS if the cache lookup is a miss, and rewrites the packet's destination MAC address field based on the ARP look-up result from the local cache or the DS.

3.2 Centralized IP Address Resolution

Because *Peregrine* is designed to scale to a large number of physical servers using only L2 connectivity, it discourages broadcast-based protocols such as ARP (Address Resolution Protocol) and DHCP (Dynamic Host Configuration Protocol). Instead, it replaces them with a client-server architecture, where queries are directed to a dedicated server, which answers these queries by looking up its internal data structures. This design change

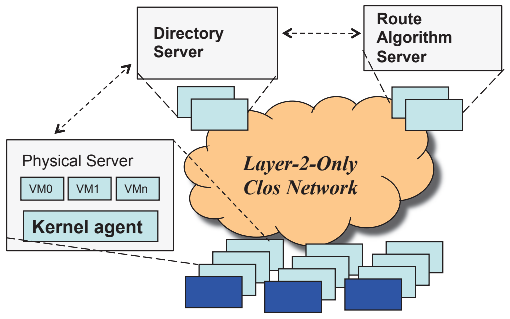


Figure 1: *The software architecture of the current* Peregrine *prototype, which consists of a kernel agent installed in the Dom0 VM of every physical machine, a centralized directory server (DS) for IP to MAC address look-up, and a centralized route algorithm server (RAS) for route computation and forwarding table population.*

is similar in spirit to how cache coherence protocols in shared-memory multiprocessor systems progressed from broadcast-based to directory-based as their scale increases.

When a user VM sends out a broadcast-based ARP query, a *Peregrine* agent running at the same physical server intercepts it, converts the query into a unicast packet and sends it to a central *directory server*, which maintains an *address resolution map* between VMs' IP addresses and their MAC addresses, and answers these transformed ARP queries. After receiving answers from the directory server, the *Peregrine* agent converts it into a legitimate ARP response packet, sends it to the original querying user VM, and caches the answers for future reuse. Therefore, not every ARP query needs to be sent to the directory server; in fact, most ARP queries are expected to be answered by the caches maintained by *Peregrine* agents.

To ensure the consistency of ARP caches, *Peregrine*'s directory server adopts a *lease-based* stateful cache coherence protocol. That is, every cached ARP query response is given a fixed lifetime, say 2 minutes, and the directory server keeps a record of which physical server caches which ARP query responses, each of which consists of an IP address and its corresponding MAC addresses (it would become clear later why multiple MAC addresses are associated with an IP address). When an ARP query response resides in a physical server longer than the fixed lifetime, it becomes invalid and cannot be used to answer ARP queries. The key design challenge in this stateful cache consistency maintenance mechanism is how to reduce the amount of state required. Suppose the maximum number of VMs in a cloud data center is 100,000 VMs, and every VM could communicate with at most *N* VMs, then the address resolution map in the directory server has 100,000 entries, each of which in turn contains up to *N* VM IDs and *N* timestamps of when the entry is cached in each of the *N* VMs. Whenever the directory server modifies an entry in its address resolution

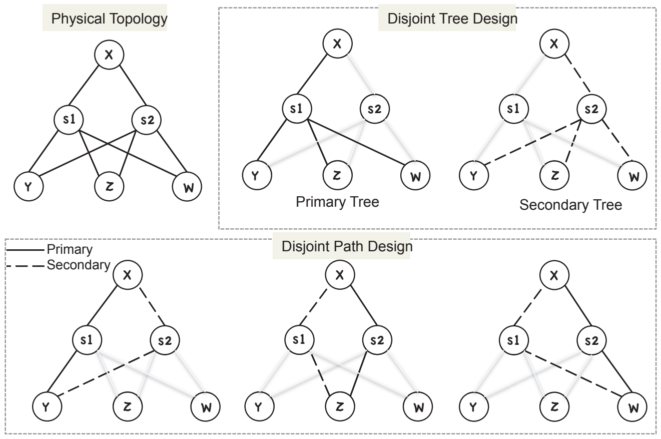


Figure 2: *An example network to illustrate the difference between a disjoint tree design (above) and a disjoint path design (below). In the disjoint path design, the primary path of one server pair (X-Z) could overlap with the secondary path of another server pair (X-Y) as long as it does not overlap with the first server pair's corresponding secondary path.*

map, it goes through the physical servers recorded in the entry, checks their timestamps to see if they expire, and sends a unicast-based invalidation notification to each of those that still hold a valid cached copy. If the number of physical servers caching an address resolution map entry exceeds *N*, the entry is flagged, and the directory server sends a broadcast-based invalidation notification instead when the entry is modified.

The additional level of indirection introduced by the address resolution map and the directory server's ability to invalidate cached ARP query responses plays a critical role in *Peregrine*, and serves multiple purposes, including scaling up the network size, redirection in VM migration, and fail-over in network switch/link failure, which we will describe in the next subsection.

3.3 Proactive Primary/Secondary Routing

When a network switch/link fails, the design goal of *Peregrine* is to route all affected network flows around the failure so that the end-to-end disruption to the communicating parties of these network flows is no more than 100 ms. To achieve this aggressive goal, for a given physical server X, *Peregrine* proactively pre-computes a *primary* and a *secondary* route from every other physical server to X, where the primary route and secondary route are node-disjoint and link-disjoint excluding the two end points, assuming the underlying physical network connectivity offers enough redundancy for such disjoint paths. Whenever a network link or switch fails, the primary routes provisioned on the failed device or link are identified, and the physical servers that are using these primary routes are notified to switch to their corresponding secondary routes. In this design, the fail-over delay of a network device/link failure thus consists of (a) the time to detect the device/link failure, (b) the time to identify affected primary routes and their source physical servers, and (c) the time to inform these affected source servers to switch from primary to secondary routes.

The first design issue is how to switch from primary to secondary routes when failures occur. Because *Peregrine* uses conventional Ethernet switches and Ethernet switches forward packets based on their destination address, the only way to send packets to a given physical server X using multiple routes is to assign multiple MAC addresses to X, each representing a distinct route to reach X. At start-up time, *Peregrine* installs pre-computed primary/secondary routes to every physical server in the switches' forwarding tables. At run time, switching from the primary to the secondary route of a given server is a matter of using the server's secondary MAC address rather than primary MAC address. Modern operating systems, including both Linux and Windows, allow multiple NICs to be bound to the same IP address, through DHCP or through user-entered commands.

Given a physical server X, all other servers that *send* packets to it form a spanning tree. When computing primary and secondary routes for X, there are two possible designs: *disjoint tree* and *disjoint path*. In the *disjoint tree* design, the system computes a primary spanning tree and a secondary spanning tree that are node-disjoint and link-disjoint from each other. In the *disjoint path* design, the system computes a primary route and a secondary route between X and *every other* server, and they are node-disjoint and link-disjoint. In the first design, all other servers that send packets to X either use the first or second spanning trees, but not both simultaneously. However, in the second design, some servers that send packets to X may use the first spanning tree, while the others may use the second spanning tree at the same time. Figure 2 shows an example that illustrates the difference between these two designs. In the disjoint tree design, the primary spanning tree rooted at node X is disjoint from X's secondary spanning tree. However, in the disjoint path design, the primary path for X-Z could overlap with the secondary path for X-Y , and the secondary path for X-W could overlap with the primary path for X-Y. Obviously the disjoint path design is more flexible than the disjoint tree design, but also requires more state to be maintained on the directory server.

The trade-off between these two designs is the amount of directory server state required and the routing flexibility. In general, the larger-granularity the unit of disjointness, the more difficult it is to successfully overlay two such units on a given physical network. Because the granularity of disjointness in the *disjoint tree* design is larger than that in the *disjoint path* design, it is more difficult to successfully compute routes for the *disjoint tree* design than for the *disjoint path* design on the same physical network. That is, for a given physical server X, it is less likely to find two disjoining spanning trees rooted at X, than to find two disjoint paths between X and any other server. In addition to routing flexibility, the *disjoint tree* design also incurs higher collateral damage when a network switch/link failure. In other words, any failure that affects (even a slight portion of) a given server's primary spanning tree renders the entire spanning tree unusable.

Given a lookup request for a physical server X, the directory server's response to it is independent of the source issuing the lookup request in the *disjoint tree* design, but is dependent on the source in the *disjoint path* design. The additional flexibility enables the *disjoint path* design to use both spanning trees associated with a physical server simultaneously, but also requires more state to be maintained on the directory server in the second design. For the *disjoint tree* design, the directory server only needs to maintain two bits for each physical server to indicate the health status of its two disjoint spanning trees. For the *disjoint path* design, the directory server needs to maintain two bits for every other server that sends packets to every given server to indicate the health status of the two disjoint paths between them. Therefore, the amount of availability-related state on the directory server is O(M) for the *disjoint tree* design and O(M^2) for the *disjoint path* design, where M is the number of physical servers on the network. As shown in Figure 2, in the disjoint tree design, the directory server maintains two bits for X's primary tree and backup tree and a failure of any link in the primary tree triggers all other servers to switch to the backup tree, whereas in the disjoint path design, the directory server maintains two-bit health status for paths between Y to X, Z to X, and W to X. If a link between X and s1 fails, only the primary path between X and Y and the secondary paths between X and Z and between X and W are affected.

The current *Peregrine* prototype adopts the *disjoint path* design to successfully fail over as many communicating node pairs affected by a given network switch/link failure as possible. To reduce the amount of availability-related state on the directory server, *Peregrine* uses a list structure that can dynamically grow and shrink its size to record the set of physical servers with which a given physical server is currently communicating. From the analysis of several data center traffic traces [9], we assume the majority of physical servers communicate with at most N other servers, and the total amount of availability state that needs to be maintained is proportional to MN rather than M^2.

The availability bits associated with a physical server are stored in the server's address resolution map entry in the directory server, together with its timestamps associated with stateful caching. In summary, a physical server's address resolution map entry consists of the following:

- An IP address,
- Two MAC addresses, and
- A communication list of entries, each of which contains a caching timestamp, two availability bits and a primary/secondary flag for each physical server that it currently communicates with.

Every physical server, say S1, is assigned an address resolution map entry, and every other physical server that communicated with S1, say S2, is assigned an entry in S1's communication list, which indicates which of S1's two MAC addresses is the primary MAC address and whether the two paths between S1 and S2 are available or not. When another server, say S3, just starts to communicate with S1, *Peregrine* inserts an entry <011> [1] to S1's communication list, meaning that the currently used MAC address is primary and both routes from S3 to S1 are available. As soon as a link on the primary route from S3 to S1 fails, S3's entry becomes <101>, indicating that S3 should use S1's second MAC address to reach S1, and the old primary route is now unavailable.

Conventional Ethernet switches use a source learning mechanism to populate their forwarding table, and thus do not support dynamic routing that could accommodate fluctuating traffic workloads. Only Layer-3 routers provide such support. Most commodity Ethernet switches provide the flexibility to statically and programmatically populate their forwarding table. *Peregrine* leverages this capability to support a centralized routing architecture, in which a route server computes the routes according to a number of optimization criteria, and populates the resulting routes on the switches' forwarding tables. *Peregrine* uses a load-based routing algorithm [7] that dynamically computes routes based on link loads. To support fast fail-over, *Peregrine* extends this algorithm to pre-compute two disjoint routes for each pair of physical servers. To support network QoS, *Peregrine* gives different weights to physical server pairs so that routes computed for higher-priority physical server pairs should travel on less congested network links than lower-priority physical server pairs.

3.4 Fast Fail-Over for Network Failure

The RAS detects a link failure by receiving an SNMP trap about it. Because a switch failure is effectively the same as multiple link failures, when a switch fails, the RAS receives one or multiple SNMP traps indicating

[1] Bit [1:0] indicates the health status of the primary and secondary path. Bit [2] indicates the selected path (0: Primary, 1: Secondary).

failures of links associated with the switch. To ascertain whether such a switch indeed fails, the RAS continuously pings the switch for a period of time (currently set to 1 second) before arriving at a verdict. Therefore, the switch failure detection time is longer than the link failure detection time.

When the RAS detects a link or switch failure, it invokes *Peregrine*'s failure recovery processing algorithm as follows:

1. Given a failed link/switch, RAS consults with the inverse map to identify all physical server pairs whose primary or secondary route traverses through the failed link/switch, and passes these physical server pairs to the DS.
2. For each physical server pair whose primary route is affected, the DS looks up the pair's destination in its address resolution map, turns off the primary route between them in the corresponding address resolution map entry, and notifies the pair's source server to this effect if the pair of servers are actively communicating.
3. For each affected physical server pair, the RAS removes the forwarding table entries associated with its affected primary or secondary route, and computes a new route for it.

Suppose a physical server S1 is affected by a link failure, and there are N_1 other servers that could send packets to S1 over the failed link, but only N_2 of them are actively communicating with S1 at the time of the link failure. So S1's address resolution map entry originally contained a list of N_2 entries to indicate S1's availability status to the N_2 servers *before* the link failure, but the list will grow to N_1 entries *after* the link failure. It is necessary to expand an affected physical server's communication list in its address resolution map entry to correctly instruct those physical servers that are not communicating with the affected server which MAC address to use when they start communicating with the affected server in the future.

Figure 3 illustrates how *Peregrine*'s fast fail-over mechanism works. Initially, VM6's primary and secondary MAC addresses, `mac1` and `mac2`, are prepopulated on the switches along the two disjoint routes by the RAS (step 1). The primary route to VM6 goes through SW2 and SW3 while the secondary route goes through SW1 and SW4. Whenever a link along the primary path from VM3 to VM6 is down, an SNMP trap is sent from the link's adjacent switch to the RAS (step 2), which determines the physical server pairs that are affected by the link failure and passes these affected server pair information to the DS (step 3), which then informs the source of each physical server pair that its associated destination server is reachable only via its secondary

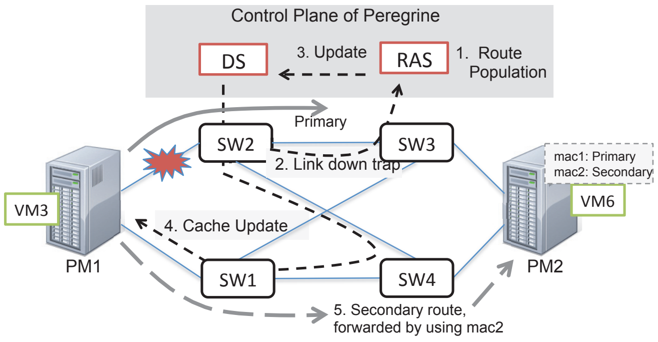


Figure 3: *When a link (e.g. PM1-SW2) fails,* Peregrine *switches every affected server pair (e.g. PM1-PM2) from its primary path (PM1-SW2-SW3-PM2) to its secondary path (PM1-SW1-SW4-PM2).*

MAC address, in this case, sending an ARP entry update to PM1 (step 4) indicating that to send packets from VM3 to VM6 should use `mac2` as the destination MAC address, the secondary MAC address for VM6. After that, all packets destined to VM6 from VM3 will go through VM6's secondary route from this point on (step 5).

3.5 Fast Fail-over for DS/RAS Failure

Because the DS and RAS play a critical role in *Peregrine*'s architecture, it is essential that both of them include a high availability (HA) mechanism to ensure their continued operation despite any single failure of their underlying hardware. First of all, all data structures in the DS and RAS that are required to restart must be kept on disk, and make up their persistent state. We adopt an active master and passive slave architecture, in which the master and slave each have their own local disk. Every update to the master's persistent state is first logged to a memory-resident log, which is synchronously replicated to the slave, and then asynchronously written to the on-disk data structure, which is synchronously replicated to the slave.

The data structures in the RAS that need to be persistent are an in-memory log of pending SNMP traps and the computed routes for every physical server pair and the inverse map that associates network links/switches with routes that traverse them. The route-related information is largely static. The data structure in the DS that needs to be persistent is the address resolution map. *Peregrine* puts RAS's and DS's persistent state in a separate disk volume, and uses DRBD (Distributed Replicated Block Device) [6] to synchronously replicate every write to the master's on-disk persistent state to the slave, and re-synchronize a new slave candidate's on-disk persistent state to the current master's. In addition, *Peregrine* uses *Pacemaker* to monitor the health of the RAS and DS processes and the servers they run on.

The slave takes over as the new master when it detects

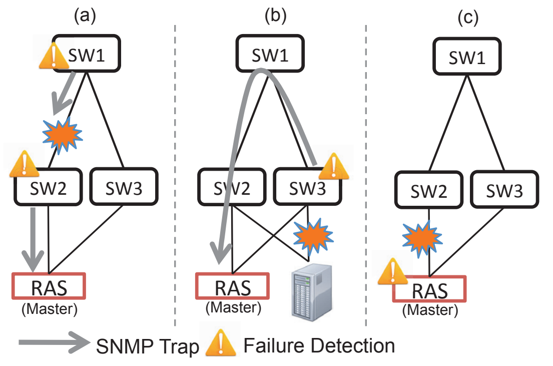


Figure 4: *Because* Peregrine *does not have an out-of-band control network,* Peregrine *must guarantee that SNMP traps be delivered to the RAS despite the failure that triggers the SNMP traps.*

the master dies. When the slave takes over, it recovers the persistent state, performs necessary undoing and redoing, and announces to the world that it is the new master. Specifically, when the slave RAS takes over, it aborts the on-going recovery processing transaction triggered by an SNMP trap if there is one pending, and redoes it from scratch. When the slave DS takes over, it rebuilds the address resolution map from the in-memory log and on-disk copy.

3.6 Resilient Messaging During Fail-over

Because *Peregrine*'s fail-over processing involves failure-detecting switches, the RAS, the DS and the affected physical servers, it is possible that a network failure prevents the communications in its associated fail-over processing and thus inhibits its own recovery. A standard solution to this problem is to install a separate control network for out-of-band fail-over processing. However, such a design is still problematic because there is no guarantee that the out-of-band control network itself won't fail. Instead, *Peregrine* uses in-band signaling to simplify the network infrastructure, but ensures the resiliency of message delivery during fail-over processing by transferring fail-over messages over paths unaffected by the triggering link/switch failures. Fail-over messages include SNMP traps, affected physical server pairs, and notifications to invalidate ARP cache entries.

The HA version of RAS consists of a master node and a slave node, and the HA version of DS also consists of a master node and a slave node. Each RAS/DS node is assigned two MAC addresses and one IP address. Because each RAS/DS node has two MAC addresses, *Peregrine* sets up two disjoint paths between it and every other node that communicates with it, and the DS decides which path should be used between each pair of communicating nodes, including the communications between the two RAS nodes and those between two DS nodes.

Every network switch is configured to send each of its SNMP traps twice, once to the master RAS and the second time to the slave RAS. Figure 4 (a) shows that if the link that fails is between two switches, both switches detect it, and at least one of them is able to send its SNMP traps to the RAS nodes in spite of the failure. If the link that fails is between a switch and a physical server, as in the case of Figure 4 (b), there is only one switch detecting the failure, and this switch definitely is able to send out the SNMP traps associated with this link failure to the RAS. If the link that fails is between a switch and a RAS node, as in Figure 4 (c), this RAS node detects this link failure itself without relying on SNMP traps. Upon receiving an SNMP trap, the master (slave) RAS synchronously replicates it to the slave (master) RAS. This replication serves two purposes: enhancing the reliability of SNMP trap delivery even when the network drops SNMP packets from time to time, and duplicating the in-memory log for RAS fail-over.

The kernel agent on every physical server constantly keeps track of the IP address and the two MAC addresses of the current master DS so that it could submit its ARP queries to the right DS node over a healthy path. In case an ARP query times out, the kernel agent retries the same query with an alternative MAC address. When a new master DS comes along, it broadcasts multiple times to announce to all physical servers its IP address and MAC addresses.

When the master RAS starts up, it establishes a UDP connection with each of the two DS nodes. Through these two UDP connections, the master RAS is able to tell which DS node is the current master DS. When a link/switch failure occurs, the DS is the one that tells every other node whether to switch paths when communicating with specific nodes, except the communication between the RAS and the DS, because this communication takes place *before* the DS is notified of the failure. Therefore, when the master RAS receives an SNMP trap associated with a link/switch failure, it first determines whether it should reach the master DS via its secondary MAC address, and informs the master DS of this failure using the pre-built UDP connection over a path unaffected by the failure. It is crucial that a UDP connection rather than a TCP connection be used here, because the return traffic (e.g. ACK packets) of a TCP connection from the master DS may be blocked by the failure. Once the master DS is informed of a failure, it adjusts its path to the master RAS to bypass the failure if necessary, and then establishes a TCP connection with the master RAS to retrieve the affected physical server pairs.

The master RAS sends the physical server pairs affected by a failure in two batches, the first batch containing those physical server pairs in which the master

DS is the source, and the second batch everything else. The master DS uses the first batch to adjust its paths to physical servers in this batch, and then sends out notifications in the second batch using the adjusted paths. To reduce the messaging overhead of notifications, the master DS further sorts the notifications according to destination nodes, and batches all notifications destined to the same node into as few packets as possible. When the per-server kernel agent receives notifications, it updates its ARP cache and its DS data structure accordingly.

3.7 Broadcast Support

Although *Peregrine* is designed to minimize broadcast traffic, it cannot completely does away with broadcast traffic. For example, ARP requests from network devices on which no *Peregrine* agent is installed, e.g. commercial routers and switches, are broadcast packets. As another example, some applications, e.g. Microsoft Exchange cluster, may use application-level broadcast messages to maintain cluster membership. To accommodate broadcast traffic and prevent Ethernet storms, the RAS sets up a tree that spans all nodes in the entire physical network without using the spanning tree protocol, and allows broadcast packets to flow only on this tree by disabling the broadcast option on all ports that are not on this tree. When a link/switch failure occurs, the RAS amends this tree accordingly to ensure the resulting tree continues to span the entire network.

4 Performance Evaluation

4.1 Evaluation Methodology

We used two racks in the ITRI container computer as the evaluation testbed for the *Peregrine* prototype. The testbed consists of eight 48-port TOR switches each with two 10GE uplink, two 48-port 10GE regional switches, and 88 physical machines. Each physical machine is equipped with eight 2.53GHz Intel Xeon CPU cores, 40GB DRAM, and 4 GE NICs, and is installed with CentOS 5.5, which is equipped with the Linux kernel 2.6.18. Four physical machines are used to deploy the RAS, DS, and their master and slave. The *Peregrine* kernel agent is installed on all other physical machines. Each physical machine is connected to four TOR switches via a separate 1GE NIC, and each TOR switch in turn is connected to four regional switches via a separate 10GE link. No firmware modifications are required on these regional or TOR switches.

To quantify the fail-over latency, we measured the service disruption time for an UDP connection running on two physical machines of the evaluation testbed under various single-failure scenarios. More concretely, the sender of this UDP connection sent one packet every millisecond to the receiver; we then counted the number of lost packets when a failure occurs and *Peregrine*'s fail-over mechanism kicks in, and the resulting number corresponds to the service disruption time.

To assess the efficiency of different fail-over steps, we broke the service disruption time into the following four components:

1. *Failure detection* time: the time between when a failure occurs and when the RAS detects the failure,
2. *Damage assessment* time: the time for the RAS to identify the set of primary and secondary routes affected by a given failure and pass the associated information to the DS,
3. *ARP Update* time: the time for the DS to update its own ARP database entries corresponding to the source nodes of affected physical server pairs and to send out ARP cache updates to the destination nodes of these pairs, and
4. *Switch-over* time: The kernel agent on a physical server updates its ARP cache upon receiving such an ARP cache update message.

To accurately measure the failure detection without installing an agent on the switches, we set up another UDP connection from a source server through the failed link or switch to the RAS, in which the source server also sends a UDP packet to the RAS every millisecond. The RAS measures the time between when it stops receiving packets through this UDP connection (a failure occurs) and when it receives the SNMP associated trap (a failure is detected). Because the *switch-over* time is negligible, we focus on the first three components in the following subsections. Each reported time measurement below is an average of 5 runs.

4.2 Link Failure

Table 1 shows the average service disruption time and its detailed breakdown for four different types of link failures: failure of a link between a server and a 1-GE switch (Server-Switch), failure of a link between a 1-GE switch and a 10-GE switch (Switch-Switch), failure of the link between the DS and a 1-GE switch (DS-Switch), and failure of the link between the RAS and a 1-GE switch (RAS-Switch). The time taken to detect a link failure and send out its associated SNMP trap is much smaller for the 10-GE switches in our testbed, between 60 ms to 80 ms, than for the 1-GE switches, between 200 ms and 1000 ms. We suspect that 10-GE switches detect the link status using event triggering scheme whereas 1-GE switches employ polling-based scheme. In the case of Switch-Switch link failures, it is a 10-GE switch that

Failed Link	No . of Affected Pairs	No. of Notifications	Failure Detection	Damage Assessment	ARP Update	Service Disruption
Server-Switch	158	8	787	13	6	810
Switch-Switch	1383	101	59	88	39	190
DS-Switch	153	73	242	34	30	300
RAS-Switch	156	134	359	29	25	420

Table 1: *The average service disruption times of four different types of link failure and their detailed breakdowns. All time measurements are in terms of ms.*

Failed Switch	No. of Affected Pairs	No. of Notifications	Failure Detection	Damage Assessment	ARP Update	Service Disruption
Regional Switch	6684	203	1881	326	234	1180
Server-Switch	3786	95	1129	156	88	1280
DS/RAS-Switch	6496	343	1407	316	223	1480

Table 2: *The average service disruption times of three different types of switch failures and their detailed breakdowns. All time measurements are in terms of ms.*

sends out the associated SNMP traps, whereas Server-Switch and DS-Switch link failures are detected by 1-GE switches. As for RAS-switch link failures, it is a kernel module in the RAS that detects them directly. As a result, in all cases except Switch-Switch, the failure detection time dominates and accounts for more than 80% of the service disruption time. Unfortunately, the failure detection time is completely determined by the switches and beyond the control of *Peregrine*.

If the failure detection time is excluded, the combined DS and RAS fail-over processing time, which is dictated by *Peregrine*, is below 120 ms for all link failures and below 70 ms if Switch-Switch link failures are ignored. The Switch-Switch link failure entails a much larger number of affected server pairs and notifications than other types of link failures. The damage assessment time is generally proportional to the number of affected server pairs (second column), and the ARP update time is generally proportional to the number of notifications that the DS sends to affected servers (third column). When the RAS sends out the list of server pairs affected by a link failure to the DS, the DS only needs to send ARP updates to destination nodes of a subset of those server pairs that are *actively* communicating with each other at that instant. That is why the number of notifications is smaller than the number of affected server pairs.

4.3 Switch Failure

Table 2 shows the average service disruption time and its detailed breakdown of three different types of switch failures: failure of a 10-GE regional switch (Regional Switch), failure of a 1-GE switch connected to a physical server (Server-Switch), and failure of the switch connected to both RAS and DS (DS/RAS-Switch). The switch failure detection time is generally higher than the link failure detection time because the RAS needs to receive multiple SNMP traps associated with link failures of a suspect switch and ping the suspect switch for 1 second without getting any response before concluding that the switch fails. The failure detection time for Regional Switch is higher than that for Server-Switch because the former is detected by 1-GE switches whereas the latter is detected by 10-GE switches. The switch failure detection time for the DS/RAS-Switch failure is about 1-second ping delay plus the link failure detection time for the RS-Switch failure, because both are detected by RAS.

For the Server-Switch and DS/RAS-Switch failure, the service disruption time for a switch failure is smaller than the sum of failure detection time, damage assessment time and ARP update time because a portion of fail-over processing is triggered by link failure SNMP traps and is thus overlapped with the switch failure detection time. The fail-over processing for those link failures whose SNMP traps cannot be successfully delivered to the RAS is triggered only after the RAS concludes that a switch failure occurs. The extent of overlap for the DS/RAS-Switch failure is higher than that for the Server-Switch failure because a significant portion of a DS/RAS-Switch failure's fail-over processing is due to the fail-over processing of the DS-Switch and RAS-Switch link failures and they are completed before the DS/RAS-Switch failure is detected. In the case of the Regional Switch failure, the service disruption time is actually smaller than the failure detection time because the fail-over processing for all the constituent link failures of a switch failure is completed before the RAS concludes that the switch failure indeed takes place.

4.4 RAS and DS Failure

When the master RAS fails, it takes on average 1038 ms for the RAS slave to notice because the RAS slave probes the RAS master for 1000 ms before declaring a take-over, and another 0.45 ms to restart itself. The restart processing of the RAS slave is fast because the only RAS persistent state is the pending SNMP trap log, which is mostly empty in this test. Because the RAS performs fail-over processing for link/switch failures, failure of the RAS master potentially increases the fail-over

Address Resolution Map Size	Service Disruption Time	Failure Detection Time	DRBD Switch Time	DS Recovery Time
6556 Entries	2811	1871	704	269
32469 Entries	3282	1882	706	736

Table 3: *The service disruption time of the DS because of a DS failure and its breakdown under two testbed sizes. All time measurements are in terms of ms.*

latency of link/switch failures. To test this, we turned off the RAS master, then immediately turned off the switch to which the RAS is connected, and measured the service disruption time of a UDP connection going through the switch. The service disruption time is increased to 1580 ms, which, as expected, is about 1000 ms higher than the average fail-over latency for link failures shown in the last subsection.

Table 3 shows the service disruption time of the DS; a DS failure is about 2811 ms and 3282 ms when the address resolution map contains 6556 entries and 32469 entries. We used a special test program that continuously submits ARP queries every 50 ms to the master DS and slave DS, and the service disruption time corresponds to the time interval between when the master DS stops responding and when the slave DS starts responding. The DRBD switch time corresponds to the time the slave DRBD needs to mount the replicated disk partition before becoming the new master DRBD. The failure detection time is bound by the Pacemaker library used in the current *Peregrine* prototype. Both the DS failure detection time and the DRBD switch time remain unchanged as the testbed size is increased. In contrast, the DS recovery time is proportional to the size of the address resolution map, because larger address resolution maps require longer reload time during recovery.

5 Conclusion

Peregrine is a software-defined network that uses commercial off-the-shelf Ethernet switches as basic building blocks and was originally designed for the ITRI container computer. It uses a centralized control plane to program the forwarding tables and configure the options of the switches in the network. Compared with conventional Ethernet architecture, *Peregrine* is more scalable because it supports dynamic load-based routing, and is more available because it provides self-adaptive fault tolerance against any single failure. More concretely, through proactive primary/secondary routing, *Peregrine* is able to significantly cut down the service disruption time due to link failures, switch failures and control plane failures. The specific research contributions of this work include

- A proactive disjoint path-based primary/secondary routing scheme that is able to quickly switch communications between server pairs affected by a link/switch failure to their pre-arranged alternative routes,
- A highly available control plane that is capable of continued operation despite any single control server failure,
- A resilient communication design that achieves reliable message transfer in fail-over processing of a link/switch failure without using any out-of-band control network, and
- A fully operational prototype that is able to cut down the service disruption time associated with any single link failure to under 120 ms, if the failure detection time is excluded.

References

[1] AWERBUCH, B., ET AL. Distributed control for paris. In *Proc. ACM PODC 2012*.

[2] BEHESHTI, N., AND ZHANG, Y. Fast failover for control traffic in software-defined networks.

[3] BOTTORFF, P., AND HADDOCK, S. Ieee 802.1 ah-provider backbone bridges, 2007.

[4] CASADO, M., ET AL. Ethane: Taking control of the enterprise. *Proc. ACM SIGCOMM 2007*.

[5] CHIUEH, T., ET AL. Peregrine: An all-layer-2 container computer network. In *Proc. IEEE Cloud, 2012*.

[6] ELLENBERG, L. Drbd 9 and device-mapper: Linux block level storage replication. In *Proc. of the 15th International Linux System Technology Conference, 2008*.

[7] GOPALAN, K., ET AL. Load balancing routing with bandwidth-delay guarantees. *Communications Magazine, IEEE, 2004*.

[8] GREENBERG, A., ET AL. A clean slate 4d approach to network control and management. *Proc. ACM SIGCOM, 2005*.

[9] GREENBERG, A., ET AL. Vl2: A scalable and flexible data center network. *Proc. ACM SIGCOMM 2009*.

[10] IANNACCONE, G., ET AL. Analysis of link failures in an ip backbone. In *Proc. ACM SIGCOMM 2002*.

[11] KOMPELLA, K., ET AL. Link bundling in mpls traffic engineering (te).

[12] KOPONEN, T., ET AL. Onix: A distributed control platform for large-scale production networks. *Proc. USENIX OSDI 2010*.

[13] MARKOPOULOU, A., ET AL. Characterization of failures in an ip backbone. In *Proc. IEEE INFOCOM 2004*.

[14] MCKEOWN, N., ET AL. Openflow: enabling innovation in campus networks. *ACM SIGCOMM 2008*.

[15] NIRANJAN MYSORE, R., ET AL. Portland: a scalable fault-tolerant layer 2 data center network fabric. *Proc. ACM SIGCOMM, 2009*.

[16] RODEHEFFER, T. L., AND SCHROEDER, M. D. *Automatic reconfiguration in Autonet*. ACM, 1991.

[17] SHARAFAT, A. R., ET AL. Mpls-te and mpls vpns with openflow. In *Proc. ACM SIGCOMM 2011*.

[18] SHERWOOD, R., ET AL. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep* (2009).

[19] STAESSENS, D., ET AL. Software defined networking: Meeting carrier grade requirements. In *Proc. IEEE LANMAN, 2011*.

[20] TAVAKOLI, A., CASADO, M., KOPONEN, T., AND SHENKER, S. Applying nox to the datacenter. *Proc. HotNets 2009*.

[21] ZHANG, Y., ET AL. On resilience of split-architecture networks. In *IEEE GLOBECOM 2011*.

Fault Management in Map-Reduce through Early Detection of Anomalous Nodes

Selvi Kadirvel, Jeffrey Ho and José A. B. Fortes
University of Florida
Email: selvik@ufl.edu, jho@cise.ufl.edu, fortes@ufl.edu

***Abstract*—Map-Reduce frameworks such as Hadoop have built-in fault-tolerance mechanisms that allow jobs to run to completion even in the presence of certain faults. However, these jobs can experience severe performance penalties under faulty conditions. In this paper, we present Fault-Managed Map-Reduce (FMR) which augments Hadoop with the functionality to mitigate job execution time penalties. FMR uses an anomaly detection algorithm based on sparse coding to anticipate a faulty slave node. This proposed technique has the following key advantages: (1) model training uses only normal-class data, (2) time taken for prediction is less than a second, and (3) confidence estimates are produced along with the anomaly prediction. FMR uses the result of anomaly detection to invoke a closed-loop recovery action, namely dynamic resource scaling. A scaling heuristic is proposed to determine the extent of scaling necessary to reduce impending performance penalty. FMR facilitates practical adoption by being implemented as a set of libraries and scripts that require no changes to the underlying source code of Hadoop. A set of realistic Map-Reduce applications were studied through a few thousand job executions on a 72-node Hadoop testbed. Detailed empirical evaluation shows that FMR successfully mitigates performance penalties from 119% down to 14%, averaged across experiments.**

I. Introduction

Innovations in infrastructure, middleware and applications have made "*big data*" analytics possible and economically viable in a wide range of fields such as bioinformatics, data mining, web indexing, document classification, recommendation systems, etc. The Map-Reduce (MR) programming paradigm [17] along with the free and widely supported open-source implementation, Hadoop [1], has become a popular choice for incorporating data analytics in industry, government as well as academic domains. One of the important benefits of this choice is that job parallelization, data distribution and fault-tolerance are facilitated and provided by the framework itself.

Enterprise data centers, in-house clusters and cloud computing environments (that host Map-Reduce platforms) experience many faults and failures as shown in recent studies by [31], [33], [32] and [3]. The causes for these faults include scale, heterogeneity, geographical distribution, configuration management over a large set of inter-dependent services and human error as illustrated in [9]. These faults adversely affect applications running in these environments resulting in job performance degradations, failed jobs, increased costs for users and loss of revenue for the provider when Service Level Objectives are violated. Wang et al. in [38], show through simulation studies that a single node fault can result in up to 139% performance slowdown in Map-Reduce. Dinu et al. in [18] record penalties of up to 350% in job run time due to TaskTracker failures. Ananthanarayanan et al. in [7], show that job completion times in Dryad (an implementation of the Map-Reduce paradigm) can be inflated by 34% because of outliers and that faster completion times (by reducing the effect of outliers) provide a competitive advantage to service providers.

These performance variabilities and penalties make it challenging to use Map-Reduce where response time is important, such as in user-facing social networking applications at Facebook [10], user-customization applications at LinkedIn [6] and user click-stream processing, web-index generation and advertisement selection applications at Microsoft [22].

Fault-managed Map-Reduce (FMR), presented in this paper, aims at mitigating these performance penalties experienced by Map-Reduce jobs. FMR uses a Monitor-Analyse-Plan-Execute (MAPE) control loop to provide an online, on-demand and closed-loop solution to fault management. In FMR, faults are anticipated through the detection of anomalous conditions that are indicative of an impending fault [31] [23].

For anomaly detection in this context, we propose the use of a simple machine-learning technique based on sparse coding. This technique satisfies the following two requirements: (1) model training using only *normal-class* data (as opposed to the use of both normal-class and anomaly-class data) and (2) fast prediction time. Normal class data captures run time behavior of a job that has not experienced a performance fault. The need for training using only normal-class data is necessary because anomaly-class data that is representative of all (or most) possible types of faults, is difficult to obtain in a production environment. Prediction computation time using the proposed sparse-coding technique is less than a second. This allows the anomaly detection module to be incorporated in an online fashion within the MAPE loop for handling faults during job execution. In addition to these essential requirements, sparse coding based anomaly detection has two other benefits. The sparse coding model is deployed locally on each slave node and does not need to communicate or synchronize with models on other nodes to make a prediction. This makes FMR applicable to both homogeneous and heterogeneous Map-Reduce environments. The time taken to train a sparse coding model is in the order of a few seconds. This makes it possible to quickly create models for a new Map-Reduce application and also to quickly re-train models when system characteristics change. Map-Reduce applications need to be instrumented to emit heart beats, which are further processed to construct feature vectors that then serve as input to the anomaly detection module.

After an anomaly is detected, FMR uses dynamic resource scaling to reduce the performance penalty due to an impending fault. A scaling heuristic is used to determine the extent of scaling necessary. This heuristic uses performance prediction models derived from our previous work [27] to estimate Map-Reduce job execution times both in fault-free and fault-present conditions. The cost due to increased execution time is compared with the cost for additional resources and then a suitable scaling decision is taken.

FMR leverages built-in features of Hadoop in order to implement its control loop. This includes features such as (1) the provision for seamless dynamic addition of slave nodes to an executing job, (2) blacklisting of slave nodes to stop assignment of new tasks to a slave node, and (3) the node health script feature to periodically monitor a user-defined set of conditions on the slave. FMR has been designed to require no changes to the underlying Hadoop code base, thereby facilitating practical adoption.

The increasing prevalence of Map-Reduce applications along with increasingly fault-prone, large-scale computing environments, makes FMR a timely and critical component to improve Map-Reduce performance in the presence of faults. The main contributions of this paper in the context of the proposed FMR are as follows:

(1) Fault anticipation and early detection through a sparse-coding based anomaly detection method. The proposed technique has a high true positive rate of 0.95 and a high true negative rate of 0.93 averaged across experiments. Additionally, it provides the benefits of short training and testing times, requiring only normal-class data for training.

(2) A closed-loop, online dynamic resource scaling approach to reduce fault-induced performance penalties. Observed performance penalties (without FMR) range between 18% up to 210%. Using FMR, penalties were brought down to values ranging between 5% to 46%. FMR has been thoroughly evaluated using a few thousand experiments on a 72-node in-house cluster. Injected faults include CPU, memory and disk hog processes as well as node crashes. Benchmark applications from the domains of text mining and machine-learning were used for the evaluation of FMR.

Other building blocks that enable FMR were proposed in our prior works: (1) a comparative evaluation of regression based machine-learning techniques for predicting the performance of Map-Reduce jobs [27] and (2) a study of the effect of various types of faults on a MapReduce job (motivating the need for FMR) and the feasibility of resource scaling to improve performance of an executing Map-Reduce job [26].

In Section II, background to the problem and related work are discussed. In Section III, the Fault-managed Map-Reduce approach is introduced. In Section IV, implementation details of FMR are described. Section V consists of experimental validation of FMR and a discussion of the results. Conclusions are presented in Section VI.

II. Background and Related Work

In this section, we summarize Map-Reduce research related to fault management and bring out the need for FMR.

Effect of faults in Hadoop: Dinu et. al [18] evaluate the behavior of Hadoop in the presence of fail-stop faults of an entire compute node as well as Hadoop components such as the TaskTracker and DataNode daemons. The authors show that TaskTracker failures can result in up to 350% penalty while DataNode failures can lead to 218% penalty. Wang et. al [38] present a Map-Reduce simulator, MRPerf and show that it can capture fault effects. Their simulation experiments show penalties up to 186% for various injected faults. Our work [26] illustrates the effect of various factors on performance penalty such as number of slave nodes, time of fault injection and fault-detection timeout interval.These research results along with the increasing importance of Map-Reduce motivates our goal for improving fault management in Hadoop.

Fault diagnosis: The Fingerpointing project, that includes works such as [30], [34], and [8] focuses on *fault diagnosis* in Map-Reduce environments. Our approach focuses on fault *detection* and fault *recovery* through an online, closed-loop approach. However, diagnosis is important and our choice of sparse coding for anomaly detection is motivated by the need to extend detection to diagnosis in order to facilitate more targeted recovery actions.

Fault handling: In Mantri [7], outliers in an executing Dryad Map-Reduce job are identified through the use of static thresholds determined from application history. The determination of the correct threshold to use is challenging and a pre-set threshold can often drift to become incorrect in dynamic environments. Hadoop provides a built-in feature called speculative execution in which slow tasks are chosen to be executed through duplicate task instances. The deficiency of speculative execution in heterogeneous environments has been addressed by the LATE algorithm proposed by Zaharia et. al [39]. FMR applies to both performance faults as well as performance faults that lead to crash faults. However, the latter condition cannot be handled by speculative execution and LATE and this is empirically illustrated in Section V. Speculative execution also uses resources inefficiently through the execution of many duplicate tasks (for e.g. in [39] it was observed that as many as 80% of tasks were speculatively executed). In contrast to speculative execution and LATE, which use progress-based analytical models for detecting a slow task, FMR uses decentralized and local machine-learning models on each node for detecting anomalies.

Performance prediction: Predicting the completion time of a MapReduce job is done through analytical models in [37] and through simulation models in [24]. In [11], the authors predict map and reduce task slowdown using the gradient boosted decision tree model. However, prediction is based on offline analysis. The anomaly detection method proposed in this paper can be used in an online fashion and hence enables incorporation into the MAPE control loop.

Anomaly detection: Tan et. al [35] propose anomaly pre-

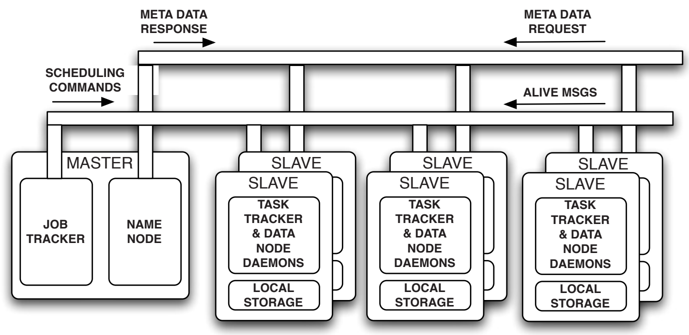
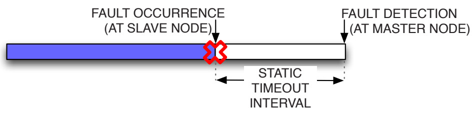
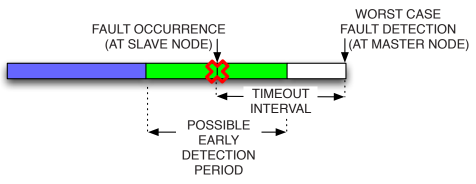


Fig. 1. Overview of Hadoop showing interactions between the JobTracker, NameNode, TaskTracker, DataNode hosted on the master and slave nodes.

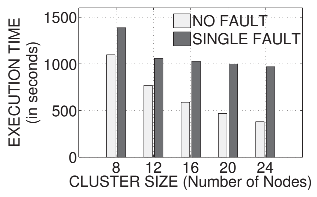

FAULT OCCURRENCE (AT SLAVE NODE)
FAULT DETECTION (AT MASTER NODE)
STATIC TIMEOUT INTERVAL

(a) Fault detection in Hadoop

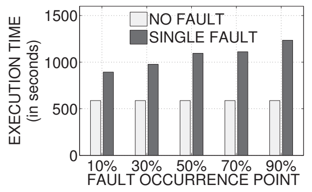

FAULT OCCURRENCE (AT SLAVE NODE)
WORST CASE FAULT DETECTION (AT MASTER NODE)
TIMEOUT INTERVAL
POSSIBLE EARLY DETECTION PERIOD

(b) Improved fault detection in Hadoop with FMR

Fig. 2. The shaded (blue) region represents map and reduce tasks running on a slave. After the fault shown by a cross tasks stop running on this slave. Hadoop master detects the fault after a static timeout value. The lightly shaded (green) region introduced into the node timeline in (b) corresponds to the period leveraged by FMR for early fault detection



(a)



(b)

Fig. 3. Execution time increase for Hadoop wordcount jobs of (a) different cluster sizes and (2) for node crash faults injected at different points in job progress.

vention schemes through the use of Markov chain models for general virtualized cloud computing. systems. However, it is a supervised learning technique which means that representative normal and anomaly class data is needed. [16] and [23] propose unsupervised techniques for anomaly detection. FMR's anomaly detection technique is similar in goal to these works.

Performance management: Starfish [24] and ARIA [37] propose the use of dynamic resource scaling for performance management of Hadoop jobs. The Starfish project does not handle faults and the ARIA project handles fail-stop faults. FMR's focus is on performance faults that result in degraded job execution times. AROMA [29] uses machine-learning techniques for resource allocation and configuration in Hadoop, however it does not handle performance deviations introduced by faults.

Our work is most similar to Jockey [22], in which resource allocation is used to guarantee job latencies for data parallel jobs. Jockey depends on an offline job profile simulator for completion time prediction; while FMR uses an online, machine-learning based model for prediction.

III. Design of Fault-Managed Map-Reduce

This paper focuses on the open-source Map-Reduce implementation, Hadoop [1]. Hadoop consists of the following main components: (1) *JobTracker* and *TaskTracker* daemons that manage scheduling and coordination of map and reduce tasks, and (2) *NameNode* and *DataNode* daemons that manage the Hadoop Distributed File Systems (HDFS). The JobTracker and NameNode daemons run on the Hadoop master node, while the TaskTracker and DataNode daemons run on the slave nodes. Figure 1 shows a simplified overview of Hadoop.

In a Map-Reduce job, when a node fails, all map tasks that were executed on this node (for this job) have to be re-executed on other healthy nodes. This is because map outputs are stored locally at each slave node (rather than being stored on the replicated HDFS). Map tasks whose outputs have already been read by corresponding reduce tasks need not be re-executed. The master node detects a slave node fault after a static timeout interval (as shown in Figure 2(a)) and then initiates re-execution. The performance penalty due to a single node fault is illustrated for Hadoop clusters of different sizes in Figure 3(a) and for the case when node faults occur at different points during a job's runtime in Figure 3(b). These penalties (ranging up to 155%) motivate the need for FMR.

One of the main contributors to the performance penalty experienced in the presence of faults is the timeout interval between fault occurrence and detection by the master. And therefore, in order to detect faults sooner, the key idea in FMR is the anticipation of a fault through anomaly detection. Figure 2(b) shows the period during which FMR attempts to detect faults. An anomaly refers to a condition that is indicative of an impending performance fault. In the context of this paper, a performance fault refers a Map-Reduce job's execution time exceeding a pre-specified Service Level Objective (SLO). By default, this SLO is the fault-free execution time. Several studies have shown that node crash faults are preceded by anomalous conditions [23] [31].

We use a machine-learning technique to identify whether a node is behaving in a manner that is unusual based on its own history for a specific type of application. After the detection of a node anomaly, recovery is initiated through dynamic resource scaling. Anomaly detection and dynamic resource scaling are described in the following subsections.

A. Anomaly Detection

Current anomaly detection techniques used in systems management depend on identifying various static thresholds as part of the control policy. When system metrics exceed these pre-determined thresholds, either alarms or suitable recovery actions are invoked. Although this simplifies the process of anomaly detection, these thresholds are difficult to determine and need to be customized as system conditions change.

In the context of machine-learning, anomaly detection can be viewed as a classification problem. Given a feature vector describing recent conditions on a compute node, we want to be able to predict whether or not this corresponds to an anomalous condition. In our work, an anomalous condition on a node could lead to a performance fault of the executing Map-Reduce job. Training data from compute nodes that are operating normally will be referred to as *normal-class* data; while those from a potentially faulty node will be referred to as *anomaly-class* data.

Application heart beats: The Map-Reduce application is instrumented to emit heart beats to indicate the rate of progress in processing input data. Heart beat timestamps are recorded locally on each slave node in a heart beat file. A sliding window of timestamps are processed to determine the heart beat rate. A sequence of heart beat rate values (referred to as a *heart beat wave*) captures application behavior and is used as the input feature vector to the anomaly detection module. Application heart beats have been used for autonomic management goals in [12] and [25].

An implicit requirement for the binary or multi-class formulation is the need for balanced and representative training data from each of the classes. In a production computational infrastructure, it is easy to obtain representative normal training data. However, requiring a system designer or administrator to provide sufficient and representative examples of anomalous data (such as from all possible performance anomalies) would strongly restrict the applicability of our approach. In order to overcome this limitation, we propose an anomaly detection method that can be trained using only normal-class data.

Sparse representation has received a great amount of attention in the signal processing community recently e.g., [20], [13], [21], and it readily provides a principled and flexible framework for feature-based anomaly detection needed in FMR. We note that there are several recent works in image processing and computer vision applying similar ideas to anomalous event detection (e.g., [15] [40]). Although originating from different application domains, these problems can all be considered as anomaly detection given only normal features. That is, anomalies are not explicitly defined based on input features but only relative to the training normal features, and this apparent asymmetry in training features is the main source of difficulty. Therefore, an algorithmic solution would require a suitable generative model for the normal features that can be used for identifying anomalies, and the sparse representation [20] offers such a model that is known for its simplicity, generalizability, and computational efficiency. Formally, in sparse representation, a feature (considered as a vector) $\mathbf{x}$ is represented as a linear combination of a small number of basis features chosen from a dictionary $\mathbb{D}$ (of basis features). In the following discussion, we will assume that $\mathbf{x}$ is a column vector of dimension d and the dictionary $\mathbb{D}$ is given as a $d \times l$ matrix such that $l > d$ ($\mathbb{D}$ has more columns than rows). The columns of the dictionary $\mathbb{D}$ are the basis features,

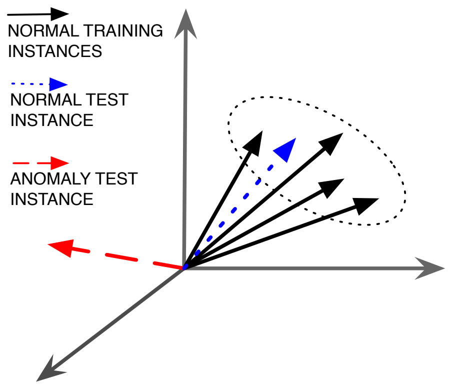


Fig. 4. A simplified diagram to illustrate the anomaly detection approach. Normal training instances (or feature vectors) are similar because they are produced by the same underlying process and hence with a high probability, lie within a confined subspace. A normal test instance is also produced by the same process and hence can be reconstructed well by other normal training instances (i.e. dictionary atoms) and as a result its sparse representation has low error values. On the other hand, an anomaly test instance is generated by a different underlying process and lies in a different subspace. Hence, when reconstructed using normal training instances, the sparse representation has larger errors.

and the main assumption in sparse signal representation is that a relevant feature $\mathbf{x}$ can be reconstructed by a small number k of columns. Mathematically, this can be written as

$$f\mathbf{x} = \mathbb{D}\,\mathbf{c_x},$$

where $\mathbf{c_x}$ is the sparse coefficient for $\mathbf{x}$ with respect to the dictionary $\mathbb{D}$, and all but k components of $\mathbf{c_x}$ are zero. The integer k is the sparsity level of the feature $\mathbf{x}$ and it tells us that the feature $\mathbf{x}$ can be reconstructed by taking a linear combination of k columns of $\mathbb{D}$. In other words, $\mathbf{x}$ is in the linear span of these k columns. Given the dictionary $\mathbb{D}$, the sparsity level k is the parameter that controls the generalizability (or expressiveness) of the model. For example, when the sparsity level is set to $k = 1$, each feature vector $\mathbf{x}$ is just a column of $\mathbb{D}$ with a scaling since we are working with $\mathbf{x}$ such that $\mathbf{c_x}$ only has one nonzero component in the equation above. For other values of k, the features are assumed to be those in the subspaces spanned by no more than k columns of $\mathbb{D}$. For this generative model, which is linear in nature, the dictionary and the sparsity level are the only two parameters used for specifying the normal features, and in particular, training of the model is exceptionally easy: simply take the normal training features as the dictionary columns. For anomaly detection, the main assumption we make in regard to the essential difference between feature vectors originating from normal operating states and anomalous states is that normal features can be sparsely approximated well using only a small number of normal features while anomaly features are expected to not enjoy this property. Therefore, given a dictionary $\mathbb{D}$ and a sparsity level k, we will consider any feature vector as a normal feature if it belongs to a subspace spanned by k columns of $\mathbb{D}$; otherwise, it will be considered as an anomaly. Figure 4 illustrates this through a simplified diagram.

More precisely, given a dictionary $\mathbb{D}$ of normal features and a sparsity level k, we expect that for a normal feature vector $\mathbf{x}$, its sparse-coding error, $\mathbf{e}$

$$\mathbf{e} = \mathbf{x} - \mathbb{D}\,\mathbf{c}_{\mathbf{x}},$$

should be a vector with small components and its magnitude follows some multivariate normal distribution (and the squared error norm $e = \|\mathbf{e}\|_2^2$ can be modeled by a χ-distribution $\phi_\chi(e)$). On the other hand, for an anomaly feature vector, its sparse-coding error $\mathbf{e}$ is expected to be large and its squared error norm does not follow the distribution ϕ_χ. Therefore, by estimating the background distribution $\phi_\chi(e)$ during training, the squared error norm e for an unknown feature vector $\mathbf{x}$ can be compared against ϕ_χ to determine its classification and the associated confidence level. We remark that the validity of using a sparse model for anomaly detection can only be supported empirically, and Figure 4 displays the results of several experiments that confirm our expectation that anomalous features incur large errors when sparsely coded with respect to the dictionary whose columns are normal features, providing a strong support for the sparse model. Furthermore, these results also suggest that the error $\mathbf{e}$ can be a useful feature for identifying the anomalies.

More specifically, the training component of our method consists of two steps: forming the dictionary $\mathbb{D}$ and estimating the distribution ϕ_χ. We randomly divide the training (normal) feature vectors into two groups. Feature vectors in the first group form the dictionary $\mathbb{D}$ and those in the second group are used to empirically estimate ϕ_χ and its cumulative distribution function $\mathbf{CDF}_{\phi_\chi}(e)$. The user specifies two parameters, $0 < \beta < \mathbf{fnr} < \alpha < 1$, which are used to bound the false negative rate $\mathbf{fnr}$ as follows: Let $e_n = \mathbf{CDF}_{\phi_\chi}^{-1}(1-\alpha)$ and $e_a = \mathbf{CDF}_{\phi_\chi}^{-1}(1-\beta)$. For any feature vector $\mathbf{x}$ with squared sparse-coding error norm e, it would be classified as normal if $e \le e_n$ or as an anomaly if $e \ge e_a$. For the "gray area" between e_n and e_a, we define the confidence level $\rho(e)$ of declaring $\mathbf{x}$ as an anomaly according to the formula,

$$\rho(e) = \frac{\mathbf{CDF}_{\phi_\chi}(e) - (1-\alpha)}{\alpha - \beta}.$$

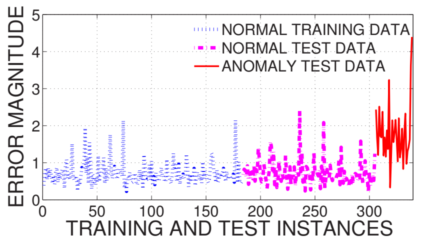
Note that $0 \le \rho(e) \le 1$ and for $\rho(e)$ to provide the confidence level, $\rho(e)$ simply scales the probability mass of ϕ_χ between e_n and e_a linearly to zero and one so that $\rho(e_n) = 0, \rho(e_a) = 1$. We also note that because we are declaring any feature vector $\mathbf{x}$ with error $e > e_a$ to be an anomaly, this gives β as a lower bound on $\mathbf{fnr}$, the false negative rate (the proportion of (training) normal feature vectors classified as anomalies). Similarly, we also have α as an upper bound for $\mathbf{fnr}$.

We remark that the key point in our method described above is the sparsity requirement, since without it, any anomaly feature vector can be approximated well using sufficiently many normal feature vectors in the dictionary $\mathbb{D}$. Only by imposing sparsity, it is then possible to use the error e as a meaningful value for classifying the feature vector $\mathbf{x}$. The sparsity requirement can further be justified using our qualitative understanding of the normal states and anomalies. In most applications, the normal features are comparatively more homogeneous than the anomalies, which due to their diverse origin and sporadic nature, are difficult to model consistently. Computationally, this homogeneity can be modeled by a dictionary $\mathbb{D}$ that captures (most of) the variability of the normal features such that each normal feature can be represented as a linear combination of only a small number (k) of basis features in $\mathbb{D}$. Therefore, this expected regularity of normal features provides the motivation and rationale for using sparse representation for their modeling. On the other hand, the heterogeneity of the anomalies precludes such modeling, and in general, an anomaly feature is not expected to be well approximated by a few basis features in $\mathbb{D}$. Therefore, using $\phi_\chi(e)$ as the background distribution, the sparse-coding error e provides a discriminative and useful quantity for classifying the feature vectors. In Figure 4, we plot these errors for three different Map-Reduce application datasets. We can observe the significant difference between errors for the normal and anomaly class instances.

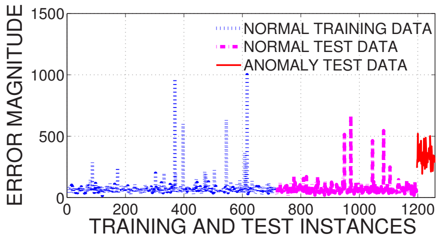
The proposed anomaly detection framework is conceptually simple and its implementation is straightforward. An important computational issue is to determine the sparse coefficients $\mathbf{c}_{\mathbf{x}}$ given a test feature $\mathbf{x}$. Fortunately, there are efficient algorithms such as orthogonal matching pursuit (MOD) [19] and LASSO [36] that compute sparse signal decomposition, given the signal $\mathbf{x}$ and dictionary $\mathbb{D}$. Using these efficient sparse coding algorithms, the running time of our method, both in training and testing, is fast and makes real-time anomaly detection feasible. Furthermore, the simplicity of our method allows various generalizations and extensions such as incorporating incremental updates of the dictionary $\mathbb{D}$ and background distribution ϕ_χ for anomaly detection in dynamic and complex environments, a topic we will pursue in the future.

B. Remediation through Dynamic Resource Scaling

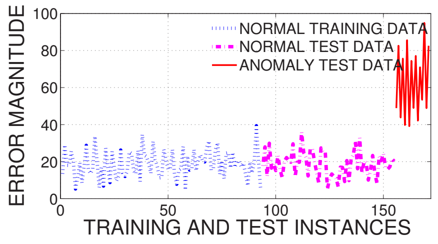
Dynamic resource scaling refers to the addition of Map-Reduce slave nodes to an executing job. This is a feasible solution to improve execution time in the presence of faults because of two reasons: (1) Hadoop allows for seamless addition of slave nodes (without restart of the master node daemons or changes to configuration files) and (2) Horizontal scaling is provided through a programming API in most virtualized and cloud environments. For a new node to be included in an executing job, TaskTracker and DataNode daemons must be started on it and the master node IP address must be provided to it. These newly started daemons will make a request for work to the master node and are then assigned data blocks to process. We note that the master node need not be aware of a slave node that may potentially be added in the future. This provides necessary flexibility to add as many nodes as needed for handling different faulty conditions. Additionally, an important design goal in our work has been to keep the underlying Hadoop framework unmodified in order to ensure that our solution can be easily adopted in practice. Dynamic resource scaling chosen as the remediation



(a) Wordcount



(b) Pi estimation



(c) Grep

Fig. 5. Errors in the sparse representation of training and test feature vectors for three Map-Reduce application datasets. Illustrates a significant difference in the magnitudes between normal class and anomaly class instances.

technique, facilitates this design goal.

The number of nodes to be added depends on the time at which the fault is detected, expected completion time of the job, the number of chunks yet to be processed and the number of nodes involved in the job. When the number of map tasks to be executed is more than the number of slave nodes available, then multiple map waves are executed. Dynamic resource scaling can help as a recovery technique for an executing job, only if at least one or more map and reduce waves are yet to be started.

The scaling heuristic that is a part of FMR needs to add sufficient number of nodes to reduce execution time penalty. Both tasks that have already completed on the faulty node and future tasks that would have executed on that node need to be executed on newly added nodes. This condition is expressed in Equation (1) where $mapProgressPercentage$ is retrieved from the Hadoop runtime using a built-in API.

$$N_{nodes\_added} = ceil\left(\frac{1}{1 - MapProgPercentage} + 1\right) \quad (1)$$

After determining the optimal number of nodes to be used for scaling (using Eq. 1), we determine the associated cost of these resources ($costOfScaling$). In order to determine whether the cost of scaling would be justified, we use Map-Reduce execution time prediction models to estimate job duration in the presence of a node fault ($execTime_{fault}$).

In our previous work [27], we have shown that Map-Reduce execution times can be estimated using machine-learning based regression models ($PerfModel$). We showed that 4 techniques, namely gaussian process regression, regression by discretization, multilayer perceptron and model trees, achieved best performance for predicting Map-reduce job execution time. Average prediction errors obtained were less than 12%. Out of these models, model trees were chosen for use in the experiments in this paper.

Using this execution time, we calculate the potential cost ($delayPenalty$) associated with exceeding the job deadline. Any user-defined cost function ($CostModel$) can be used here. The cost for the execution time penalty is compared with the cost for resource scaling, and scaling is invoked if it provides a cost benefit. This functionality of FMR is described as pseudocode in Figure 6.

```
1: $execTime_{nofault} = PerfModel(NumFaults = 0)$
2: $execTime_{fault} = PerfModel(NumFaults = 1)$
3: $delayPenalty = CostModel(execTime_{nofault}, execTime_{fault})$
4: $N_{nodes\_added} = ceil\left(\frac{1}{1-MapProgPercentage} + 1\right)$
5: $costOfScaling = nodesToScale * costPerNode$
6: if $costOfScaling < delayPenalty$ then
7:     Invoke scaling operation
8: end if
```

Fig. 6. Pseudocode of the scaling heuristic in FMR

IV. Implementation of Fault-Managed Map-Reduce

The various components of FMR that together constitute the MAPE control loop are illustrated in Figure 7 and described in this section.

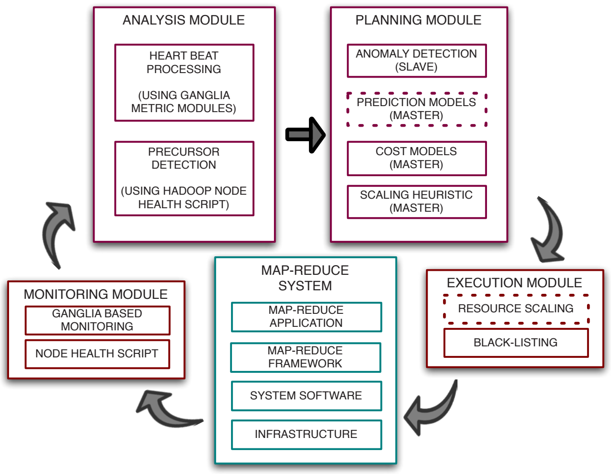


Fig. 7. Autonomic control loop of Fault-managed Map-Reduce showing a high-level overview of the monitoring, analysis, planning and execution modules. Contributions described in this paper are shown within solid-outline blocks. Contributions from prior work that are used in FMR are shown within dashed-outline blocks.

Monitoring using Ganglia: Ganglia is an open-source project [2] that provides a flexible monitoring framework for distributed systems. In FMR, customized metrics are added to Ganglia for calculating the heart beat rate and for performing anomaly detection.

Node Health Script: The node health script is a feature provided in Hadoop that allows for a pre-defined health script to be periodically executed on each slave node. As soon as a node anomaly is conveyed to the master through Ganglia, the node is *black-listed* by FMR. Black-listing immediately prevents any more tasks from being scheduled on that node. Typically, slave node faults are detected by the master only after a timeout interval. The advantage of blacklisting is that the master is made to become aware of a slave's degrading health status immediately. This is beneficial since a recovery action can be invoked without any delay. We also configure the node health script to check for other fault precursors such as task and daemon crash faults. This precursor detection functionality helps detect some crash faults that are not preceded by anomalous conditions (that could be detected by the sparse coding technique).

Anomaly Detection: The anomaly detection module is invoked at the end of each task. The input feature vector to the anomaly detection module is a heart beat wave that corresponds to the last-completed task. The sparse coding technique is implemented in Matlab and converted to a standalone executable which is then executed on the slaves using the Matlab Compiler Runtime (MCR) environment.

Recovery through Dynamic Resource Scaling: Once an anomaly is detected, the anomalous node is forcefully blacklisted. Then the scaling heuristic is executed to determine the number of nodes to be added. We use a cost model in which dollar costs are associated with different penalty ranges. Virtual machine images for the new slaves are pre-set with the master IP address. TaskTracker and DataNode daemons are started up on the new nodes, which then become a part of the executing Map-Reduce job.

V. Experimental Evaluation

Experimental Testbed: The test bed used to evaluate the FMR approach consists of 16 IBM blade servers (HS22) mounted on two different racks. Each physical node has a 8-Core Xeon 2.4 GHz CPU and 24 GB of RAM and runs CentOS 5.5 with Xen 3.4.3. The two racks are linked together by a Gigabit Ethernet network. Each physical node hosts five guest virtual machines. This guest VM (which forms a Hadoop slave node) runs Ubuntu 10.04.2 and is configured with a single core and 2Gb of RAM. Hadoop version 0.20.203 is used.

Map-Reduce applications: Applications from the Hadoop distribution and the PUMA benchmark suite [4] were used and are described below:

1) Wordcount (WC): Map outputs a *(word, 1)* key-value pair for each word in a document. Reduce combines the count for each word producing a *(word, wordcount)* pair.
2) Grep (GR): Map searches for a pattern in the input documents and produces *(pattern,1)*. Reduce combines the count for each pattern producing *(pattern, patterncount)*.
3) Pi estimation (PI): Estimates the value of Pi using quasi-Monte Carlo method.
4) Inverted index (II): Map generates the document index for each word as *(word, document index)*. Reduce combines all occurrences of a word to produce *(word, list of document indices)*.
5) Term vector per host (TV): Determines frequently occurring words in a document. Map produces *(host, termvector)* for each host. Reduce combines term vectors for each host and outputs *(host, list(termvector))*.
6) Histogram ratings (HR): Generates a histogram of movie ratings from a dataset of user reviews. Map produces *(rating, 1)* for each user review. Reduce combines the count to produce *(rating, count)*.

Input dataset: Dataset used for WC, GR, II and TV consists of books from Gutenberg [5] with size varying between 5GB to 20GB. PI does not require any input data. Input for HR is generated using scripts from PUMA.

Job duration: Performance penalties are low for long-running jobs that execute on a large number of nodes. However, long running jobs are not the common case for Hadoop as seen from two production traces that were analyzed in [14], [28]. In these studies, the average length of a job varies between few tens of seconds to few tens of minutes. The average Map-Reduce job size at Google [17] varied between 395 to 874 seconds over a period of three years between 2004 and 2007. FMR and its evaluation experiments thus focus on short jobs with runtimes ranging between 300 to 600 seconds which correspond to the majority workload in production environments.

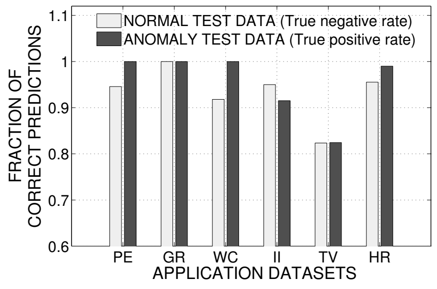
Faultload: The following fault conditions were injected into slave nodes:

1) CPU hog: A CPU-intensive sequence of matrix multiplication operations.
2) Memory hog: A sequence of memory leaks programmed into an executing matrix multiplication process.
3) Disk hog: The linux *dd* command used to copy large chunks of data between two disk partitions.
4) Node crash fault: The linux *kill* command used to terminate the TaskTracker and DataNode daemon processes running on the node.

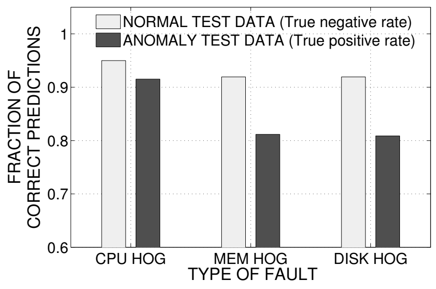
Each fault experiment consists of loading HDFS with the input, starting FMR scripts and the Map-Reduce job and then injecting faults at pre-specified time instances. Node crash faults are preceded by performance faults. After each fault experiment, HDFS is reformatted and reloaded with input data. This ensures that any non-uniformity in data distribution and replication is eliminated for each new experiment. A set of 3000 job executions were performed for validating anomaly detection, performance prediction and resource scaling components of FMR and are described in the following subsections.

A. Anomaly Detection

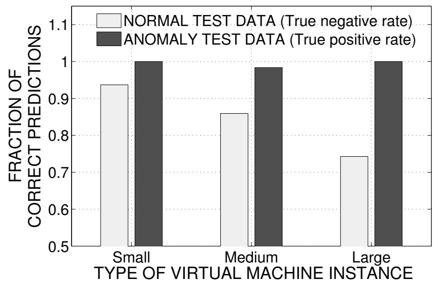
The experiment shown in Figure 8 is used to illustrate the operation of the anomaly detection module. A Wordcount Map-Reduce job is executed with a CPU hog injected into one node. We note that the anomalous slave node 'Dom-13' (in the fourth plot from the top) was correctly identified. In accordance to the goal of *early* fault detection, the fault was detected at the end of the first application heart beat wave and is marked in using an arrowhead.



(a) Different Map-Reduce applications



(b) Different types of faults



(c) Different virtual machine instances

Fig. 9. Evaluation of sparse coding based anomaly detection for (1) different Map-Reduce applications, (b) different faulty conditions, and (c) different VM instance sizes in a heterogeneous environment.

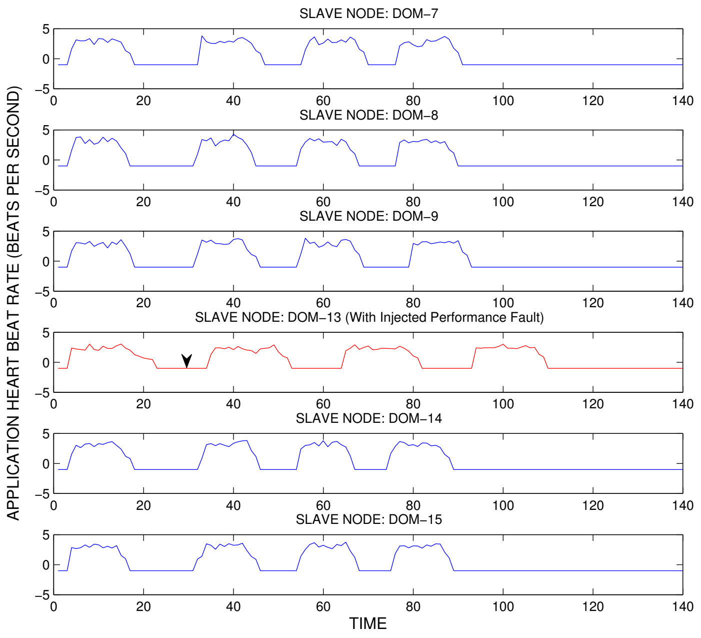
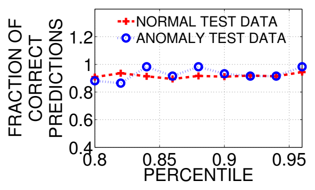
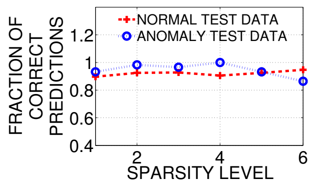


Fig. 8. Application heart beat waves for a 6-node Hadoop job in which a CPU hog process was injected into one slave node ‘Dom 13’. The anomalous node is shown in the fourth plot (from the top). An arrow head marks the time of detection of the anomaly on this node.



(a)



(b)

Fig. 10. Sensitivity analysis: Variation of the fraction of correct predictions for normal test data (true negative rate) and anomaly test data (true positive rate) using the sparse coding anomaly detection technique for different values of the (a) ‘Percentile’ parameter and (a) ‘Sparsity Level’ parameter. Application: Term vector per host. Injected fault: CPU hog.

Figure 9(a) shows the fraction of correct predictions for the normal and anomaly test data for six datasets corresponding to six different Map-Reduce applications. The fraction of correct predictions in the anomaly dataset is the the True Positive Rate (TPR) $= \frac{TP}{FN+TP}$; while the fraction of correction predictions in the normal dataset is the True Negative Rate (TNR) $= \frac{TN}{TN+TP}$. Here TP, TN, FP, FN stand for True Positives, True Negatives, False Positives and False Negatives respectively. In this context correctly detecting an anomaly is termed a True Positive. We see that for all the datasets the TNR is greater than the theoretical bound of 0.8 that was chosen for the percentile parameter. This corresponds to a maximum False Positive Rate of 0.2.

In Figure 9(b), we plot the TPR and TNR for the inverted index application for different injected faults. In order to identify the cause for variation in performance, we compare the intensity of the effect (performance penalty) of each these faults. A CPU hog, memory hog and disk hog causes 22%, 13% and 11% increase in average execution time. The CPU hog with maximum penalty is a more severe fault and hence can be detected with better performance. The memory hog and disk hog effects are more subtle and hence appear to result in slightly lesser performance.

The time taken for prediction computation and model training is shown in Table I for 3 application datasets. We see that for the largest dataset of 1400 instances, training takes 1 sec and testing takes only 0.008 secs. This ensures minimal overhead when our anomaly detection model is used within FMR.

TABLE I
TRAINING AND TEST DURATIONS FOR ANOMALY DETECTION

Dataset	Application	Total Instances	Training Duration	Testing Duration
1	Pi Estimation	1497	1.06 secs	0.008 secs
2	Grep	202	0.13 secs	0.008 secs
3	Wordcount	400	0.22 secs	0.008 secs

Heterogeneity: The use of decentralized, local models for anomaly detection enables us to extend FMR to work in a heterogenous environment. A heterogeneous testbed was configured consisting of three different virtual machine instance types: ‘small’ VMs with 1 CPU and 2GB of RAM, ‘medium’ VMs with 2 CPUs and 4GB RAM and ‘large’ VMs with 4CPUs and 6GB of RAM. Performance of anomaly detection for each VM instance type in this environment is shown in Figure 9(c).

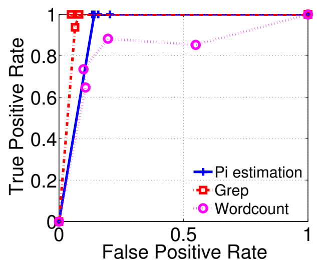
Sensitivity Analysis: In order to determine the effect of choosing different parameters, we perform a sensitivity analysis of two parameters, namely the sparsity level in Figure 10(a) and percentile value in Figure 10(b). Sparsity level is varied between 1 and 6 and the percentile parameter (which is related to $1 - \beta$) is varied between 0.8 and 0.96. The effect of these variations on TPR and TNR is plotted. We see that anomaly detection performance is quite stable within these ranges, thereby providing sufficient leeway in choosing good parameter values. We note that although anomaly data is not needed for training, it can be leveraged when available for parameter tuning.

Receiver Operating Characteristic curves: We plot ROC curves for 6 applications in Figure V-A by varying the confidence threshold between 0 and 1. All curves are close to the upper-left corner, where TPR is high and FPR is low. In addition, most of the curves provide a number of possible values of confidence threshold (i.e. points on the curve with markers) in the upper left corner region indicating that good performance is possible for many confidence threshold values.

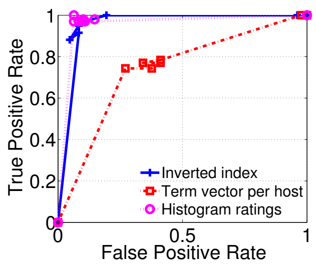
TABLE II
COMPARISON OF ANOMALY DETECTION TECHNIQUES

Application	Multilayer Perceptron	K-means clustering	Support Vector Machines	Sparse-coding
	True positive rate / True negative rate			
PI	1.0/0.96	0.99/0.88	0.76/0.3	1.0/0.93
GR	1.0/0.94	0.7/0.31	1.0/0.65	1.0/1.0
WC	0.98/0.88	0.96/1	1.0/0.69	1.0/0.92
II	0.99/0.95	0.8/0.66	1.0/0.49	0.92/0.95
TV	0.95/0.76	0.93/0.69	0.89/0.49	0.82/0.82
HR	0.99/0.98	0.975/0.99	1.0/0.51	0.99/0.96

Comparison of anomaly detection techniques: We compare sparse coding with 3 classification techniques in Table II. Multilayer perceptron provides best performance. However, it needs anomaly data for training and takes ten to a hundred seconds for training, thus making in unsuitable for FMR. K-means clustering also needs anomaly data for training and does not achieve very good TPR and TNR values. Single-class SVM does not need anomaly data for training, making it a viable candidate. However, sparse coding models achieve much better performance for all 6 benchmark applications.



(a)



(b)

Fig. 11. ROC curves showing performance of anomaly detection as the confidence threshold is varied. Six benchmark applications are shown in 2 separate plots (a) and (b) for clarity.

B. Dynamic Resource Scaling

We first evaluate the accuracy of model tree prediction, which is a critical component of the FMR control loop. Prediction accuracy for 6 test jobs is shown in Figure 12 and was an average of 11.1%. Features used include number of

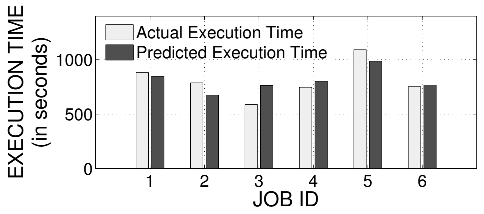


Fig. 12. Prediction accuracy of the model tree algorithm used for Map-Reduce job execution time prediction. Application: Wordcount. Jobs 1, 2, 3 have one fault injected, while jobs 4, 5 and 6 are fault-free.

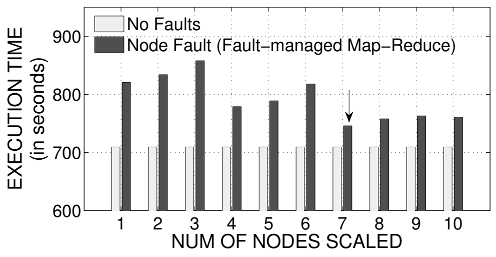


Fig. 13. Evaluation of FMR scaling heuristic. Application: Pi estimation. Fault: Node crash fault at 520 seconds. FMR scaling heuristic scales by 7 nodes (marked with an arrow in the plot)

slaves, dataset size, time of fault, number of faults, timeout and Hadoop framework configuration parameters. The model tree implementation in the Weka tool suite was used for training and testing.

Next we evaluate the scaling heuristic in Figure 13 to determine whether sufficient number of nodes are chosen for scaling. For the job shown, the heuristic scales by 7 nodes based on the map progress percentage of 0.83 in Eq (1). We manually scale by 1 to 6 nodes and 8 to 10 nodes, to determine if 7 is the right choice. We see that for $N_{nodes\_added} < 7$, penalty is $> 5\%$ and for $N_{nodes\_added} > 7$ there is no additional benefit.

The penalty reduction through dynamic resource scaling is illustrated using a *swimlane* plot in Figure 14. In this plot, each y-axis coordinate corresponds to the execution of a map task in a single map-slot. Our experiments use Hadoop's default setting of two slots per node. So a pair of consecutive lines (parallel to the x-axis) correspond to two map tasks running simultaneously on a node.

Figure 14 (a) shows a Map-Reduce job that consists of four map waves. The job did not experience any faults. In Figure 14 (b), the same job is rerun with a CPU performance fault injected into one of the nodes. The presence of a fault results in an execution time penalty of 18.5%. In the next execution of the same job, FMR scripts are enabled. Figure 14 (c) shows the addition of two nodes to the running job after detection of the anomalous node. We note that with the help of resource scaling, performance penalty is reduced to 4.6%.

In Figure 15, FMR is compared with Hadoop's built-in speculative execution. After a job begins execution, a CPU hog process is injected and is followed by a node crash fault after 30 seconds. We see that with speculative execution, penalty is not reduced. However using the FMR approach, through

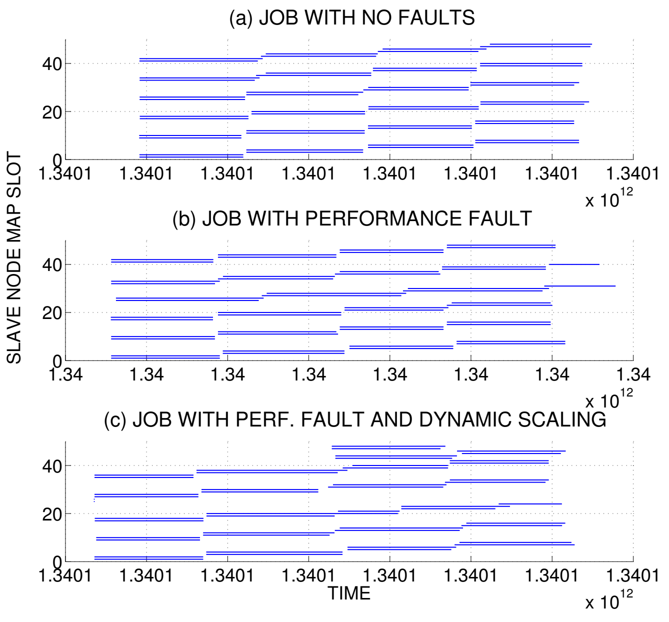


Fig. 14. Swimlane plots of three Map-Reduce jobs. (a) Job with no faults (b) Job with injected CPU performance fault (Execution time penalty of 18.5%) and (c) Job with injected performance fault and dynamic resource scaling enabled (Execution time penalty reduced to 4.6%).

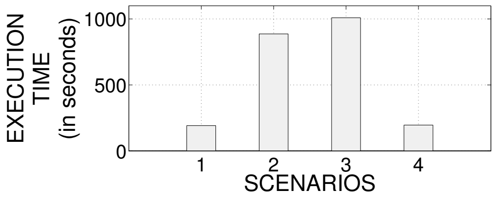


Fig. 15. Comparison of FMR with speculative execution. (1) Job with no faults (2) Job with fault and speculative execution turned off. (3) Job with fault and speculative execution turned on. (4) Job with faults managed using FMR approach. Application: Inverted Index. Fault: CPU hog process + node crash fault after 30 seconds.

scaling by 4 nodes, the penalty is decreased to $< 5\%$ of the fault-free execution time.

In the next set of experiments, node crash faults are injected into a running job executing on different numbers of nodes. As shown in Figures 16(a) to 16(f), we see that FMR consistently helps in mitigating performance penalty. FMR helped decrease performance penalty from an average of 119% to 14% across these 6 sets of experiments.

C. Discussion

Virtualized environment: FMR has been implemented on a virtualized environment which easily provides the actuators needed for dynamic resource scaling. However, a non-virtualized environment can also use FMR by provisioning extra resources that can be added to the Hadoop cluster on-demand. These extra resources can be utilized for executing preempt-able jobs during those periods when they are not utilized as part of recovery. However, a virtualized environment provides the capability to extend recovery to other actions such as migration (to handle hardware faults) and scaling up (to handle resource exhaustion faults).

Application-specific anomaly detection models: The anomaly detection model developed is specific to a Map-Reduce application because each application has different heartbeat characteristics. We believe that it is reasonable to manage application-specific models because typical Map-Reduce workloads involve the execution of the same job on gradually evolving data sets. Recent literature shows that 80% of jobs in a workload from Yahoo! were repeated at least 50 times [11]. The feature vectors needed for training the sparse coding model are derived from one application heart beat wave that corresponds to the processing of one data chunk by a map task. Hence, even a single MapReduce job can provide few tens to a few hundred feature vectors for training.

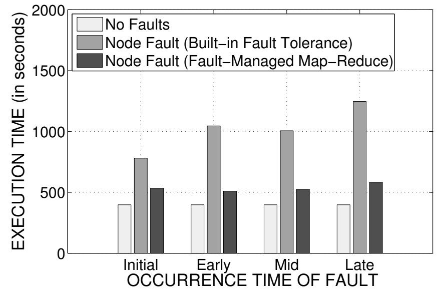
Scalability of FMR: Since anomaly detection is performed by local, decentralized models at a node, the associated overheads are local to a node. Therefore, the overhead does not increase adversely as the number of slaves is increased. The computation performed at the master (by FMR) for each slave is limited and only consists of evaluating two binary metrics for each slave (namely the presence/absence of an anomaly and the result of the node-health script). The latest version of Hadoop, called 'NextGen' Hadoop uses distributed masters and will further help reduce time taken for this evaluation. Furthermore, FMR aims at achieving soft deadlines for Map-Reduce jobs. In a typical shared Map-Reduce cluster only a subset of the jobs would have these soft-deadline requirements. Thus, FMR needs to be enabled only for these jobs, thus avoiding the need to monitor and manage all jobs.

VI. Conclusions

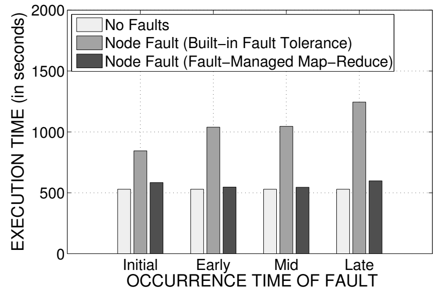
Map-Reduce has become an important platform for a variety of data processing applications. Built-in fault-tolerance mechanisms in Map-Reduce frameworks such as Hadoop, suffer from performance degradations in the presence of faults. Fault-managed Map-Reduce, proposed in this paper provides an *online*, *on-demand* and *closed-loop* solution to managing these faults. The control loop in FMR mitigates performance penalties through early detection of anomalous conditions on slave nodes. Anomaly detection is performed through a novel sparse-coding based method that achieves high true positive and true negative rates and can be trained using only normal class (or anomaly-free) data. The local, decentralized nature of the sparse-coding models ensures minimal computational overhead and enables usage in both homogeneous and heterogenous Map-Reduce environments. After an anomalous condition is detected, dynamic resource scaling, through the proposed scaling heuristic, is invoked as the recovery action. Through extensive evaluation of a variety of benchmark applications on a 72-node Hadoop cluster, we show that FMR can effectively mitigate performance penalties.

VII. Acknowledgements

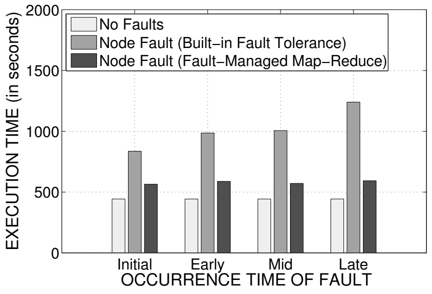
The authors would like to thank Dr. Nagarajan Kandasamy for helping shepherd our paper. This work is supported in part by the National Science Foundation grants No. OCI-0910812 and No. 0758596 and the AT&T Foundation.



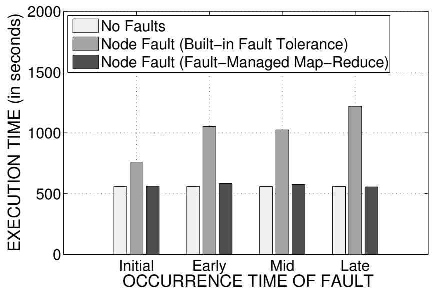
(a) 12 node cluster + 10 nodes for scaling



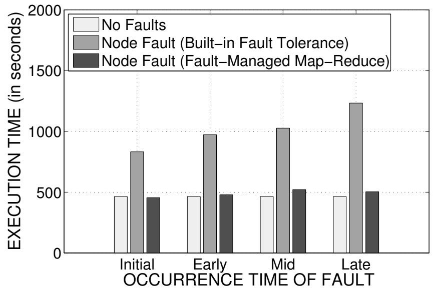
(b) 22 node cluster + 10 nodes for scaling



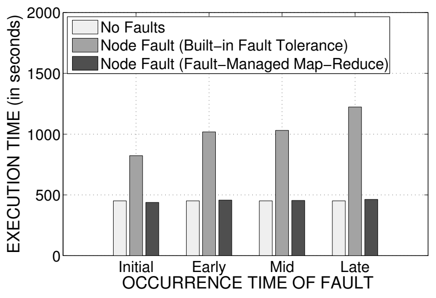
(c) 32 node cluster + 10 nodes for scaling



(d) 42 node cluster + 10 nodes for scaling



(e) 52 node cluster + 10 nodes for scaling



(f) 62 node cluster + 10 nodes for scaling

Fig. 16. Comparison of job execution times of a Pi Estimation job in the presence of a node crash fault through the use of Hadoop's built-in fault tolerance and FMR. x-axis labels 'Initial', 'Early', 'Mid' and 'Late' correspond to a node crash fault injected at 1 sec, 120 sec, 220 sec and 320 sec from job start.

REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Ganglia Monitoring Tool. http://ganglia.sourceforge.net/.

[3] Jeff Dean. http://tinyurl.com/87kgcev.

[4] Purdue MapReduce Benchmark Suite. http://tinyurl.com/bn5gmga.

[5] Gutenberg. http://www.gutenberg.org/, 2009.

[6] Jay Kreps. http://tinyurl.com/cu24pwz, 2009.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, 2010.

[8] K. Bare, S. P. Kavulya, J. Tan, X. Pan, E. Marinelli, M. Kasick, R. Gandhi, and P. Narasimhan. Asdf: an automated, online framework for diagnosing performance problems. 2010.

[9] L. A. Barroso and U. Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[10] D. Borthakur and et al. Apache hadoop goes realtime at facebook. In *2011 ACM SIGMOD Intl. Conf. on Management of Data*, 2011.

[11] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *HotCloud*, 2012.

[12] E. S. Buneci and D. A. Reed. Analysis of application heartbeats: learning structural and temporal features in time series data for identification of performance problems. In *Proc. of Supercomputing*, 2008.

[13] E. Candes and T. Tao. Decoding by linear programming, 2004.

[14] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011.

[15] Y. Cong, J. Yuan, and J. Liu. Sparse reconstruction cost for abnormal event detection. In *Proc. of CVPR*, 2011.

[16] X. G. Daniel Dean, Hiep Nguyen. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proc. of ICAC*, 2012.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] F. Dinu and T. E. Ng. Understanding the effects and implications of compute node related failures in hadoop. In *Proc. of HPDC*, 2012.

[19] D. L. Donoho, Y. Tsaig, I. Drori, and J. luc Starck. Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit. Technical report, 2006.

[20] M. Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.

[21] Y. C. Eldar and M. Mishali. Robust recovery of signals from a structured union of subspaces. *IEEE Trans. Inf. Theor.*, 55(11), 2009.

[22] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[23] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjorner. Latent fault detection in large scale services. In *Proc. of DSN*, 2012.

[24] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC*, 2011.

[25] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application Heartbeats. In *Proc. of ICAC*, 2010.

[26] S. Kadirvel and J. Fortes. Towards self-caring mapreduce: Proactively reducing fault-induced execution-time penalties. In *HPCS*, 2011.

[27] S. Kadirvel and J. Fortes. Grey-box approach for performance prediction in map-reduce based platforms. In *Proc. of ICCCN*, 2012.

[28] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *CCGRID*, 2010.

[29] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, 2012.

[30] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: BlackBox diagnosis of MapReduce systems. *SIGMETRICS Perform. Eval. Rev.*, 37(3), Jan. 2010.

[31] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of FAST*, 2007.

[32] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance comp. systems. *Trans. on Dep. and Sec. Comp.*, 2010.

[33] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. of FAST*, 2007.

[34] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan. Kahuna: Problem diagnosis for Mapreduce-based cloud computing environments. In *Proc. of NOMS*, 2010.

[35] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proc. of ICDCS*, 2012.

[36] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[37] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce with performance goals. In *Middleware*, 2011.

[38] G. Wang, A. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, 2009.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of OSDI*, 2008.

[40] B. Zhao, L. Fei-Fei, and E. P. Xing. Online detection of unusual events in videos via dynamic sparse coding. In *Proc. of CVPR*, 2011.

Reliability and Timeliness Analysis of Fault-tolerant Distributed Publish / Subscribe Systems

Thadpong Pongthawornkamol[*]
University of Illinois
Urbana, IL
tpongth2@illinoisalumni.org

Klara Nahrstedt
University of Illinois
Urbana, IL
klara@illinois.edu

Guijun Wang
Boeing Research & Technology
Seattle, WA
guijun.wang@boeing.com

Abstract

Distributed publish / subscribe paradigm is a powerful data dissemination paradigm that offers both scalability and flexibility for time-sensitive applications. However, its nature of high expressiveness makes it difficult to analyze or predict the performance of publish / subscribe systems such as event delivery probability and end-to-end delivery delay, especially when the publish / subscribe systems are deployed over distributed, large-scale networks. While several fault tolerance techniques to increase reliability in distributed publish / subscribe systems have been proposed, event delivery probability and timeliness of publish / subscribe systems with such reliability enhancement techniques have not yet been analyzed. This paper proposes a generic model that abstracts the basic distributed publish / subscribe protocol, along with several commonly used fault-tolerant techniques, on top of distributed, large-scale networks. The overall goal of this model is to predict quality of service (QoS) in terms of event delivery probability and timeliness based on statistical attributes of each component in the distributed publish / subscribe systems. The evaluation results via extensive simulations with parameters computed from real-world traces verifies the correctness of the proposed prediction model. The proposed prediction model can be used as a building block for automatic QoS control in distributed, time-sensitive publish / subscribe systems such as subscriber admission control, broker deployment, and reliability optimization.

1 Introduction

Recently, distributed publish / subscribe systems have emerged as an effective multi-source, multi-sink communication paradigm for large-scale, time-sensitive applications such as stock report, live sportcasting, and social network messaging. The main concept of publish / subscribe paradigm is that senders and receivers of information are connected loosely based on the content of the information. Specifically, in a publish / subscribe system, a *publisher* (i.e., sender) can produce its events (i.e. *messages*) without specifying the set of *subscribers* (i.e., receivers). Instead, each subscriber specifies the content (or topic) of event it is interested to receive. All events produced from publishers containing content (or topic) that match a subscriber's interest are then delivered to that subscriber via a network of intermediary servers called *brokers*. Since the information flows based on the content of the information, publishers and subscribers are decoupled in space, time, and synchronization [16]. Such transparency allows the system to scale and adapt well under dynamic environments, resulting in wide adoption of publish / subscribe paradigm in many contexts such as cloud computing [26], mobile computing [9], and peer-to-peer services [36].

While distributed publish / subscribe systems achieve scalability and fault tolerance, failures at brokers or links between brokers can still cause time-sensitive events to be lost or expired before being delivered to the subscribers [19, 28]. Several reports have shown that while high-end commercial servers with high maintenance generally achieve at least 99.9% availability (i.e., available 99.9% of the time) [4], most standard, off-the-shelf commodity servers with low to moderate maintenance may have less than 90% availability [3]. To cope with such component failures in the context of publish / subscribe systems, several fault-tolerant / fault-recovery techniques have been proposed to increase service availability in distributed publish / subscribe systems [13, 14]. However, to the best of our knowledge, the effect of such commonly-used fault-tolerant techniques to a publish / subscribe system's reliability and timeliness has not been analyzed yet.

In this paper, we propose a quantitative, analytical model to predict the effect of failures and commonly used recovery techniques to the quality of service (QoS) each subscriber receives in distributed, time-sensitive publish / subscribe systems. The primary goal of such analytical model is to estimate each subscriber's *real-time re-*

[*]The author is currently employed at Google Inc.

liability, which is the percentage of events that are successfully delivered to each subscriber on time (i.e., before the event is expired). The analytical model covers common component failures and recovery mechanisms, resulting in the model's high applicability. The evaluation results via simulations with parameters computed from real-world traces yield correctness of the proposed analytical model. The proposed model can be used as a building block for automatic QoS control in distributed, time-sensitive publish / subscribe systems such as subscriber admission control, broker deployment, and reliability optimization.

The organization of this paper is as follows. In Section 2, we first describe the basic publish / subscribe model, failure model, and commonly used fault tolerance / recovery techniques. In Section 3, we then formulate the subscriber real-time reliability estimation problem and propose a generic, protocol-independent analytical framework to solve such problem. In Section 4, we propose a set of protocol-dependent, publisher-subscriber pairwise reliability estimation models for each fault tolerance / recovery mechanism from Section 2. Section 5 then presents the evaluation results to validate the proposed analytical model via simulations. Related work is summarized in Section 6. Conclusions and future work are then discussed in Section 7.

2 Model and Assumptions

In this section, we describe the details of the distributed publish / subscribe architecture model, quality of service (QoS) model, failure model, and commonly used fault-tolerant techniques used in this paper. In the next section, we will then present the mathematical framework that realizes the model described in this section in order to predict QoS level each subscriber receives.

2.1 Distributed Publish / Subscribe Model

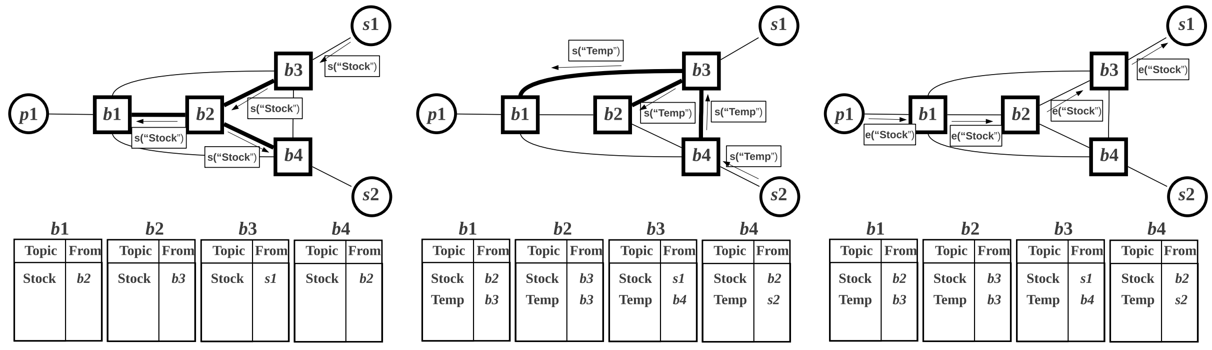
We model the publish / subscribe network as a network of brokers. Brokers can be placed inside the same domain (e.g, brokers within cloud), across different private domains (e.g., federated clouds), or across different public domains (e.g., peer-to-peer systems). Each subscriber / publisher is connected to only one of the brokers. The broker that is connected to a subscriber / publisher is called the *home broker* with respect to that subscriber / publisher. Figure 1 shows an example of a publish / subscribe network with 4 brokers , where the home brokers of subscriber s_1, subscriber s_2, and publisher p_1 are broker b_3, b_4, and b_1 respectively.

When a subscriber joins the system, it chooses a broker in the system as its home broker and sends its *subscription* message to the home broker. The subscription message specifies a *topic*[1] value, which describes the category of event that the subscriber wants to receive. Upon receiving the subscription from the subscriber, the home broker stores the subscription and the subscriber into its routing table before propagating the subscription message to other brokers. Each published event has *topic* and *deadline* associated with it. When a publisher publishes a new event, the publisher sends the published event to its home broker, who then routes the event via the broker network to all subscribers whose subscriptions have the same topic as the published event. If the event arrives at a subscriber before the event's deadline, we say that the event is delivered to that subscriber on time. Otherwise, we consider that event to be *expired* with respect to that subscriber.

For genericity, this paper does not make any assumption about subscription propagation / event routing processes within broker network. The only assumptions are that the broker network must be *stable* (i.e., neighborhood relationships between brokers do not change frequently over time) and the event routing path must be *consistent* (i.e., for each publisher-subscriber pair, brokers will always use the same path to route all events from the publisher to the subscriber). For demonstration, this paper focuses on *tree-based forwarding* (e.g., Figure 1), which is a publish / subscribe routing scheme that satisfies path consistency assumption. In tree-based forwarding scheme, a broker tree overlay is arbitrarily but consistently formed for each topic. When a broker receives a new subscription, the broker stores the subscription and its source to the broker's routing table before forwarding the subscription to its neighbors in the subscription's topic broker tree. Figure 1(a) shows an example when subscriber s_1 subscribes to topic "Stock", which forms the broker tree rooted at broker b_2. Note that subscriptions with different topics can have different corresponding broker trees. For example in Figure 1(b), subscriber s_2 subscribes to topic "Temp", which forms the broker tree rooted at broker b_3. Upon receiving a published event from a publisher, the publisher's home broker checks the event with each subscription stored in its routing table. For each subscription whose topic matches the event, the broker forwards the event to the link which it receives that subscription from. Note that an event is forwarded *once* per link even though there are multiple matching subscriptions from that link. The event forwarding process then continues, and the event is propagated hop-by-hop along the topic tree in the reverse direction of the subscription until it reaches the designated subscribers. Figure 1(c) shows an example of publisher p_1 publishing an event with topic "Stock".

While Figure 1 describes topic-based, tree-based pub-

[1]While this paper focuses on topic-based publish / subscribe, our approach can be extended to support content-based publish / subscribe as well [29].



(a) s_1 subscribes with topic 'Stock' (b) s_2 subscribes with topic 'Temp' (c) p_1 publishes an event with topic 'Stock'

Figure 1: Example of subscription propagation and event routing in a tree-based publish / subscribe network

lish / subscribe model, our analytical model can be applied to content-based publish / subscribe systems with any routing scheme that satisfies the path consistency assumption [29].

2.2 QoS Model

In our previous work, we proposed the subscriber-level QoS metric called *subscriber real-time reliability* [30]. Subscriber real-time reliability can be defined as follows: A subscriber s is said to receive the service with real-time reliability $r_s \in [0,1]$, where r_s is defined as the fraction of all events of s's interest that arrives at s before its deadline (i.e., delivery delay less than the message lifetime). This metric both standard reliability and timeliness properties, making it a suitable indicator of QoS in time-sensitive publish / subscribe applications.

2.3 Failure Model

In this paper, we discuss two types of failures that could affect subscriber real-time reliability : *link failures* and *broker failures*. We assume crash-recovery failure model for both broker failures and link failures, which means each broker / link is assumed to be either *on* or *off* at any point of time. When a broker fails (i.e., is in *off* state), it stops its activity until it recovers (i.e., is repaired back to *on* state). When a failed broker recovers, it loses all of its soft-state information (e.g., subscription routing table and queued events) that it had before the failure. However, we assume that the failed broker does not lose the broker graph information (i.e., the list of all brokers and their neighborhood relationship), which is stored in each broker's persistent, non-volatile storage. When a link fails, any event that is sent to the link will be lost.

In this paper, we assume that link failures and broker failures are *independent* and *exponentially distributed* for analysis feasibility. Previous studies also have shown that the assumption of exponential time between failures is true in many distributed systems [3,35].

2.4 Fault-Tolerant Mechanisms

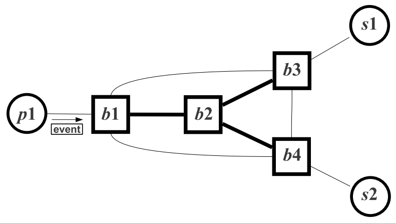
In order to cope with broker and link failures, several fault tolerance / recovery schemes for publish / subscribe systems have been proposed [7,8,13,19,22]. This section summarizes and discusses such techniques. Note that some of these techniques are generic and not limited to publish / subscribe systems. However, this paper focuses on the analysis of such techniques in the publish / subscribe context.

2.4.1 Periodic Subscription

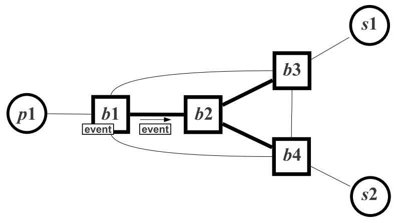
In periodic subscription scheme [33], each subscriber periodically re-issues its subscription message to its home broker, which then propagates the subscription to other brokers in the network. Each broker also maintains a timestamp for each subscription entry in its routing table. The timestamp is refreshed every time the broker receives the corresponding subscription. The broker discards any subscription from its routing table if the subscription is not refreshed within a period of time (i.e., timeout). The periodic subscription scheme can help prevent subscription loss, but it cannot prevent event loss. More details about periodic subscription can be found by several previous works [19,20].

2.4.2 Event Buffering / Retransmission

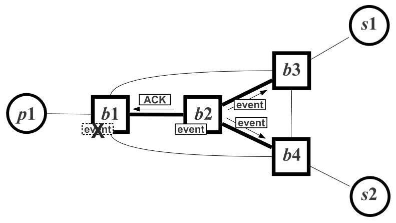
In event buffering scheme [8, 13], each broker ensures event delivery to its next hop neighbor as follows. When a broker receives an event from one of its immediate neighbors, it performs the event matching and calculates the event's forwarding set (i.e., the set of immediate neighbors to forward the event to). The broker then stores the event and its forwarding set into the broker's non-volatile storage and sends the acknowledgment message (ACK) containing the event sequence number back to its upstream neighbor. The broker then forwards the event to the event's forwarding set. The broker then waits



(a) Publisher publishes an event



(b) Home broker stores and forwards event



(c) Broker receives ACK and discards event

Figure 2: Example of event buffering / retransmission scheme

for the ACK message from each next-hop neighbor in the event's forwarding set. The broker discards the event from its non-volatile storage once it collects all the ACK messages from all brokers in the forward set, as now it is certain that the event has been received by all of the next-hop brokers. If, due to failures, the broker does not receive ACK messages from some next-hop neighbors, it retransmits the event to each of such neighbors until all ACK messages are collected or the buffered event becomes expired. Figure 2 illustrates an example of event buffering / retransmission scheme.

The event buffering / retransmission guarantees that the events will not be lost due to broker / link failures. However, if the routing path is disconnected for too long, the event may be expired before it is delivered to the subscribers.

2.4.3 Redundant Path Bypassing

Redundant path bypassing scheme relies on the fact that even when the routing path between a publisher to a subscriber is disconnected due to broker / link failures, it might be possible to find another publisher-subscriber path that excludes the failed brokers / links [8,19,22,23].

The detail of the redundant path bypassing in the context of tree-based publish / subscribe is as follows[2]. Whenever a broker detects a change of its neighbor's state (e.g., neighbor fails, neighbor recovers), it uses a link state protocol to broadcast the update message to all other reachable brokers. Each broker can update the global view of the entire broker network. With the up-to-date global view of the network, each broker can identify the set of *immediately reachable children* of a failed broker along the tree. The immediately reachable children of a broker b is the set of b's next-hop brokers that are available and reachable. For example in Figure 3, the immediately reachable children set of failed broker $b2$ are $b3$ and $b4$. Hence, each event that is supposed to be sent to the failed broker will be forwarded to the failed broker's immediately reachable children instead (i.e., detour

[2]The path bypassing technique can be used with other routing schemes as well.

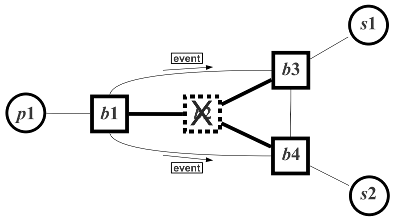


Figure 3: Example of path bypassing scheme where broker $b1$ bypasses failed broker $b2$

routing). The same approach applies to subscription forwarding as well.

The goal of this paper is to estimate each subscriber's real-time reliability when each of such fault-tolerant techniques is employed in the publish / subscribe system. Section 3 will present the generic estimation framework while Section 4 will present the estimation algorithm for each specific fault-tolerant technique.

3 Subscriber Real-time Reliability Estimation Framework

In this section, we propose a mathematical model of the publish / subscribe system and use the proposed model to formulate the subscriber real-time reliability estimation problem. We then present the generic estimation algorithm to solve the problem.

3.1 Problem Formulation

We present the mathematical model of each component in the publish / subscribe system as follows.

3.1.1 Real-time Event Model

Let $\mathbb{E}$ be the set of all events ever published in the system. Each event $e \in \mathbb{E}$ contains topic τ_e and lifetime d_e, which is the duration since the time the event was published until the time the event is expired. Without loss of generality, we assume that all events in the system have the same lifetime D (i.e., $\forall e \in \mathbb{E} : d_e = D$), which is a known constant. Our scheme can be modified to work when events have different lifetime values as well.

3.1.2 Publisher / Subscriber Model

Let $\mathbb{P}$ and $\mathbb{S}$ be the set of all publishers / subscribers in the system. Each publisher $p \in \mathbb{P}$ publishes events with topic τ_p with rate λ_p. Each subscriber $s \in \mathbb{S}$ is interested in events with topic τ_s. A subscriber s is said to be a recipient of a publisher p if they share the same topic of interest (i.e., $\tau_s = \tau_p$). We assume that λ_p, τ_p, and τ_s are known for all publishers and subscribers.

3.1.3 Broker / Link Model

Let $\mathbb{B}$ be the set of all brokers in the system. We model each broker $b \in \mathbb{B}$ as a tuple (γ_b, σ_b), where γ_b and σ_b are exponentially distributed failure rate and repair rate, respectively. That is, the broker b has exponentially distributed time between failures and time to repair with mean $\frac{1}{\gamma_b}$ and $\frac{1}{\sigma_b}$, respectively.

Hence, broker b's availability, denoted by a_b, is the fraction of time the broker b is on, which can be computed as $a_b = \frac{\frac{1}{\gamma_b}}{\frac{1}{\gamma_b}+\frac{1}{\sigma_b}}$.

Likewise, let $\mathbb{L}$ be the set of all links in the system. Each link $l \in \mathbb{L}$ has exponentially distributed time between failure and time to repair with rate γ_l and σ_l respectively[3]. Link l's availability value a_l is also calculated as $a_l = \frac{\frac{1}{\gamma_l}}{\frac{1}{\gamma_l}+\frac{1}{\sigma_l}}$

We assume that each broker's failure / repair rates (γ_b, σ_b) and each link's failure / repair rates (γ_l, σ_l) are known via statistical history data collection [1,17,21].

With the described mathematical model, we formulate the subscriber real-time reliability estimation as follows.

Subscriber Real-time Reliability Estimation Problem : Given a publish / subscribe overlay network $\mathbb{G} = (\mathbb{N}, \mathbb{L})$ where $\mathbb{N} = \mathbb{B} \cup \mathbb{P} \cup \mathbb{S}$, estimate the value of subscriber real-time reliability r_s for each subscriber $s \in \mathbb{S}$.

We use the term r'_s to denote the estimated value of subscriber real-time reliability r_s. The goal of our analytical model is to calculate r'_s that estimates r_s as accurately as possible (i.e., $\min|r'_s - r_s|$).

[3]Without loss of generality, we assume that the local link connected between publish / subscriber to its home broker does not fail. Our scheme can be simply modified for non-reliable local link scenarios.

3.2 Generic Estimation Framework

This section describes a generic framework to estimate subscriber real-time reliability. The subscriber real-time reliability estimation problem can be generally broken down into two sub-problems, which are estimating publisher-subscriber pairwise flow rate and estimating publisher-subscriber pairwise flow reliability.

3.2.1 Pairwise Flow Rate

The publisher-subscriber pairwise flow rate is the average event traffic flow rate from a publisher to a subscriber when no failure occurs. The publisher-subscriber pairwise flow rate λ_{ps} between a publisher $p \in \mathbb{P}$ and a subscriber $s \in \mathbb{S}$ can be calculated as follows.

$$\lambda_{ps} = \begin{cases} \lambda_p & \text{if } \tau_p = \tau_s \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

That is, the pairwise traffic flow rate from publisher p to subscriber s is equal to publisher p's publishing rate if they share the same topic, and equal to zero otherwise. Since λ_p, τ_p, and τ_s are known for each publisher $p \in \mathbb{P}$ and subscriber $s \in \mathbb{S}$ in the system, we can calculate pairwise traffic flow rate λ_{ps} for each publisher-subscriber pair in the system.

3.2.2 Pairwise Reliability

The publisher-subscriber pairwise reliability is the probability that a publisher's event of a subscriber's interest will be delivered to that subscriber before it is expired. We use the notation $r'_{ps} \in [0,1]$ to denote the pairwise reliability between publisher $p \in \mathbb{P}$ and subscriber $s \in \mathbb{S}$. As mentioned, the pairwise reliability depends on the fault-tolerant technique used in the publish / subscribe system. Section 4 presents the calculation of pairwise reliability for each technique discussed in Section 2.4.

3.2.3 Generic Estimation Algorithm

Subscriber s's real-time reliability r_s is the probability that s will receive an event of its interest successfully before the deadline D. Hence, the estimated value of r_s, denoted by r_s can be calculated as the weighted average of publisher-subscriber pairwise reliability between each publisher to that subscriber, with the weight equal to the pairwise event flow rate from the corresponding publisher to that subscriber. That is,

$$\begin{aligned} r'_s &= \frac{\text{E[rate of events delivered on time to } s \text{]}}{\text{E[total rate of events of } s\text{'s interest]}} \\ &= \frac{\sum_{p \in \mathbb{P}} (r'_{ps} \cdot \lambda_{ps})}{\sum_{p \in \mathbb{P}} \lambda_{ps}} \end{aligned} \tag{2}$$

Equation (2) is a generic equation that can be used with any fault-tolerant mechanism discussed in Section 2.4.

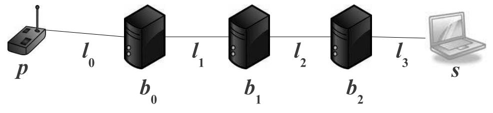


Figure 4: Example of a publisher-subscriber path with length 3

However, different mechanisms require different equations to estimate pairwise reliability r'_{ps}, which will be demonstrated in the next Section.

4 Pairwise Reliability Estimation

This section proposes an analytical model to calculate the estimated publisher-subscriber pairwise reliability r'_{ps} for each different fault tolerance/recovery protocols presented in Section 2.4.

4.1 Static Path

Without any reliable mechanism, the subscription information stored at each broker about the subscriber will be eventually lost when that broker fails. If a subscriber does not have reliable subscription or periodic subscription mechanisms, its subscription along the routing path will be eventually lost, preventing any subsequently published event to be delivered to the subscriber. Hence, the steady-state pairwise reliability each publisher-subscriber pair (p,s) will be zero (i.e., $r'_{ps} = 0$).

4.2 Static Path + Periodic Subscription

With the use of periodic subscription (Section 2.4.1), each failed broker can recover its routing information once it recovers from its failure. However, event loss is still possible as there is no reliable acknowledgement.

We analyze the pairwise reliability of each publisher-subscriber pair with static path as follows. Let δ_{ps} denote the event routing path between a publisher p and a subscriber s. Since we assume our publish / subscribe routing scheme to be consistent, δ_{ps} is static and known for each publisher-subscriber pair (p,s). We define *path length*, denoted by $|\delta_{ps}|$, as the number of brokers in the path. Hence, a path δ_{ps} can be expressed as the sequence $(p,l_0,b_0,l_1,b_1,...,l_{|\delta_{ps}|-1},b_{|\delta_{ps}|-1},l_{|\delta_{ps}|},s)$. Figure 4 shows an example of a path of length 3.

Since the static path scheme always uses only one path δ_{ps} to forward events between publisher-subscriber pair (p,s), the pairwise real-time reliablity r'_{ps} is equal to the fraction of time that the path δ_{ps} is connected. That is,

$$\begin{aligned} r'_{ps} &= \text{P[path } \delta_{ps} \text{ is connected]} \\ &= a_{b_0}\Pi_{i=1}^{|\delta_{ps}|-1}(a_{l_i}\cdot a_{b_i}) \qquad (3) \end{aligned}$$

where a_x is the availability of component (broker or link) x.

4.3 Static Path + Event Buffering

With the reliable acknowledgment protocol (i.e., Section 2.4.2) in static path, an event of a subscriber s's interest that is published by a publisher p will be eventually delivered to s, given that p's home broker is available when p publishes the event (since we assume that p does not have retransmission capability). This is because the event will always be buffered at some broker along the path between p and s, even when the path is disconnected[4]. The event will then be forwarded when the next-hop broker and link are available, and eventually delivered to the subscriber. However, the delay the event spends in the buffer may be longer than its lifetime, which causes the event to be expired.

To analyze the pairwise reliability r'_{ps} between a publisher p and a subscriber s under static path with event buffering scheme, consider the single, unique path δ_{ps} connecting p and s. Assuming the event arrival time to be independent from the path δ_{ps}'s state, we estimate the path real-time reliability as follows.

$$\begin{aligned} r'_{ps} &= \text{P[an event from } p \text{ arrives at } s \text{ on time]} \\ &= \text{P[}p\text{'s home broker is on].} \\ &\quad \text{P[end-to-end delay less than event lifetime]} \\ &= a_{b_0}\cdot \text{P}[d_{ps} < D] \qquad (4) \end{aligned}$$

where d_{ps} is the end-to-end delivery delay and D is the event lifetime. Thus, it is necessary to calculate the end-to-end delivery delay distribution d_{ps} first in order to estimate path reliability r'_{ps}.

To calculate delay distribution d_{ps} for path δ_{ps}, we need to calculate per-hop buffering delay at each broker $b_i (0 \le i < |\delta_{ps}|)$ in the path (we assume that link transmission delay and broker processing delay are negligible compared to buffering delay). Consider when the event is received successfully at broker b_i and hence broker b_i will try to forward the event to broker b_{i+1}. If both link l_{i+1} and broker b_{i+1} are up at the moment, the event will be transmitted successfully to broker b_{i+1} immediately, thus incurring zero buffering delay at broker b_i. However, if either link l_{i+1} or broker b_{i+1} is down at the moment, the event will be buffered at the broker b_i, which will keep retransmitting the event until the event gets through to broker b_{i+1}. The broker b_i discards the event if the event expires. Note that the event will get through only when all b_i, l_{i+1}, and b_{i+1} are up at the same time.

Let d_{b_i} be the buffering delay at each broker b_i $(0 \le i < |\delta_{ps}|)$. We first calculate the probability that $d_{b_i} = 0$ (i.e., the probability that the event is successfully delivered to b_{i+1} immediately), which can be calculated as

[4]In the analysis, we assume each broker to have unbounded buffer such that it can always store any incoming event.

$$
\begin{aligned}
\mathrm{P}[d_{b_i}=0] &= \mathrm{P}[b_i, l_{i+1}, b_{i+1} \text{ are on} | b_i \text{ is on}] \\
&= \mathrm{P}[l_{i+1}, b_{i+1} \text{ are on}] \\
&= a_{l_{i+1}} \cdot a_{b_{i+1}} \qquad (5)
\end{aligned}
$$

Given that delay is always non-negative, we have

$$
\begin{aligned}
\mathrm{P}[d_{b_i} > 0] &= 1 - \mathrm{P}[d_{b_i}=0] \\
&= 1 - a_{l_{i+1}} \cdot a_{b_{i+1}} \qquad (6)
\end{aligned}
$$

Now, in the case that the buffering delay at each broker b_i is not zero (with probability $1 - a_{l_{i+1}}.a_{b_{i+1}}$), we need to find the delay distribution in such case. Let $d^+_{b_i}$ be the conditional buffering delay at broker b_i under the condition that $d_{b_i} > 0$. Assuming the event arrives at arbitrary time at broker b_i, the conditional buffering delay $d^+_{b_i}$ is equal to the time it takes for the next-hop path to be repaired (i.e., time until l_{i+1} and b_{i+1} are both in *on* state). Assuming each component's time between failure and time to repair to be exponentially distributed, we can calculate such delay distribution by using continuous-time Markov process diagram that represents the state of broker b_i, link l_{i+1}, and b_{i+1}. The diagram is shown in Figure 5. Each of 8 states depicts each possible state of sub-path (b_i,l_{i+1},b_{i+1}), with each bit representing each individual component's state (0 = *off*, 1 = *on*). The first bit (least significant bit) represents b_i's state. The second bit represents l_{i+1}'s state. The third bit (most significant bit) represent b_{i+1}'s state. For example, state "011" represents the state where broker b_i is on, link l_{i+1} is on, and broker b_{i+1} is off. Note that in the scenario where an event arrives at broker b_i and needs to be buffered at b_i, an event will find the system state in either state "001", "011", or "101" with probability $\frac{(1-a_{l_{i+1}})(1-a_{b_{i+1}})}{1-a_{l_{i+1}}.a_{b_{i+1}}}$, $\frac{a_{l_{i+1}}(1-a_{b_{i+1}})}{1-a_{l_{i+1}}.a_{b_{i+1}}}$, and $\frac{(1-a_{l_{i+1}})a_{b_{i+1}}}{1-a_{l_{i+1}}.a_{b_{i+1}}}$ respectively. The event will continue to be buffered at broker b_i (note that b_i can also fail but the event is kept in its non-volatile storage) until the state becomes "111", which the event will be transmitted to broker b_{i+1} successfully. Hence, the diagram depicts the absorbing Markov process with three start states = "001", "011", "101" and one absorbing state "111"with the corresponding transition rate matrix $\dot{Q}$ as

$$
\dot{Q} = \begin{pmatrix}
-\dot{q}_0 & \sigma_{b_i} & \sigma_{l_{i+1}} & 0 & \sigma_{b_{i+1}} & 0 & 0 & 0 \\
\gamma_{b_i} & -\dot{q}_1 & 0 & \sigma_{l_{i+1}} & 0 & \sigma_{b_{i+1}} & 0 & 0 \\
\gamma_{l_{i+1}} & 0 & -\dot{q}_2 & \sigma_{b_i} & 0 & 0 & \sigma_{b_{i+1}} & 0 \\
0 & \gamma_{l_{i+1}} & \gamma_{b_i} & -\dot{q}_3 & 0 & 0 & 0 & \sigma_{b_{i+1}} \\
\gamma_{b_{i+1}} & 0 & 0 & 0 & -\dot{q}_4 & \sigma_{b_i} & \sigma_{l_{i+1}} & 0 \\
0 & \gamma_{b_{i+1}} & 0 & 0 & \gamma_{b_i} & -\dot{q}_5 & 0 & \sigma_{l_{i+1}} \\
0 & 0 & \gamma_{b_{i+1}} & 0 & \gamma_{l_{i+1}} & 0 & -\dot{q}_6 & \sigma_{b_i} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} \qquad (7)
$$

where γ_x and σ_x are component x's exponential failure rate and exponential repair rate described in Section 3.1, and $\dot{q}_i$ is state i's total outgoing rate. For example, $\dot{q}_0 = (\sigma_{b_i} + \sigma_{l_i} + \sigma_{b_{i+1}})$. Thus, the conditional buffering delay at broker b_i is equal to the time to absorption of the absorbing matrix $\dot{Q}$, which is a phase-type distri-

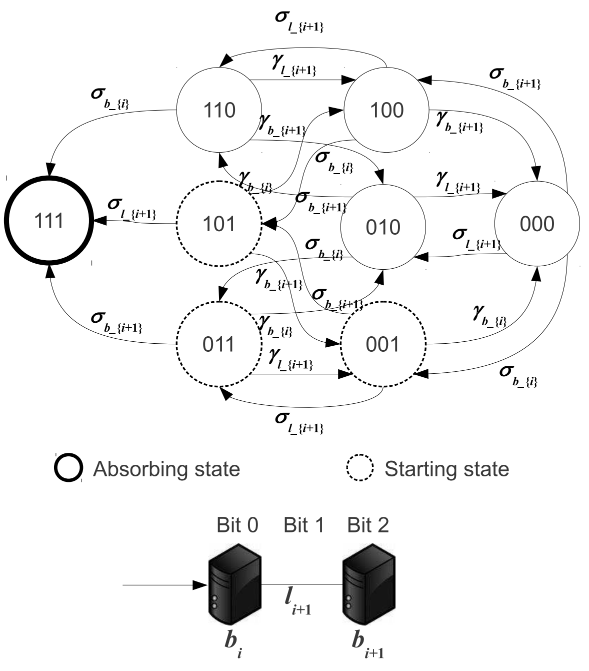


Figure 5: 8-state continuous, absorbing Markov process diagram for per-hop buffering delay analysis

bution [25] and can be calculated by breaking down the matrix $\dot{Q}$ in to the form of

$$
\dot{Q} = \left(\begin{array}{c|c} \dot{S} & \dot{\mathbf{S}}^0 \\ \hline \mathbf{0} & 0 \end{array} \right) \qquad (8)
$$

Where $\dot{S}$ and $\dot{\mathbf{S}}^0$ are the 7x7 top-left sub-matrix and the 7x1 top-right sub-vector of $\dot{Q}$ defined in Equation (7) respectively. Hence, the cumulative distribution of $d^+_{b_i}$ can be calculated as

$$
\mathrm{P}[d^+_{b_i} < t] = 1 - \alpha \cdot \exp(\dot{S}t) \cdot \mathbf{1} \qquad (9)
$$

where $\exp(\dot{S})$ is the matrix exponential [27] of $\dot{S}$, α is the 1x7 starting state vector

$$
\alpha = [0, \frac{(1-a_{l_{i+1}})(1-a_{b_{i+1}})}{1-a_{l_{i+1}}.a_{b_{i+1}}}, 0, \frac{a_{l_{i+1}}(1-a_{b_{i+1}})}{1-a_{l_{i+1}}.a_{b_{i+1}}}, 0, \frac{(1-a_{l_{i+1}})a_{b_{i+1}}}{1-a_{l_{i+1}}.a_{b_{i+1}}}, 0]
$$

and $\mathbf{1}$ is an 7x1 vector with every element being 1.

With Equation (9), we can compute the conditional buffering delay distribution $d^+_{b_i}$ at broker b_i. Hence, we can estimate buffering delay d_{b_i} at broker b_i as

$$
d_{b_i} = \begin{cases} d^+_{b_i} & \text{with probability } 1 - a_{l_{i+1}} \cdot a_{b_{i+1}} \\ 0 & \text{with probability } a_{l_{i+1}} \cdot a_{b_{i+1}} \end{cases} \qquad (10)
$$

Once we calculate per-hop buffering delay d_{b_i} with Equation (10), we then can calculate the end-to-end buffering delay d_{ps} for path

$\delta_{ps} = (p, l_0, b_0, l_1, b_1, ..., l_{|\delta_{ps}|-1}, b_{|\delta_{ps}|-1}, l_{|\delta_{ps}|}, s)$ as

$$d_{ps} = \sum_{i=0}^{|\delta_{ps}|-1} d_{b_i} \tag{11}$$

Hence, Equation (11) completes the calculation of pairwise reliability for static path with event buffering scheme in Equation (4).

4.4 Path Bypassing + Periodic Subscription

With the path bypassing scheme discussed in Section 2.4.3, a new routing path between p and s will be used if the old path fails. Hence, an event of a subscriber s's interest that is published by a publisher p will be delivered to s as long as the broker graph $\mathbb{G}$ is not partitioned between p and s. Thus, pairwise reliability r'_{ps} is then equal to the graph $\mathbb{G}$'s connection probability between p and s. However, the calculation of such connection probability for any generic graph is considered to be a #P-complete problem [34], which has higher complexity that a NP-complete problem.

Due to such computational complexity, we propose an algorithm to approximate the lower bound of graph $\mathbb{G}$'s connection probability between any publisher-subscriber pair (p, s) by constructing a subgraph $\mathbb{G}' \subseteq \mathbb{G}$ that consists only parallel, broker-disjoint paths between p's home broker and s's home broker.

That is, the multi-path subgraph $\mathbb{G}'$ contains multiple, broker-disjoint path between p's home broker and s's home broker, assuming there are m of such paths in subgraph $\mathbb{G}'$, namely $\delta_{ps}^{(0)}, \delta_{ps}^{(1)}, ..., \delta_{ps}^{(m-1)}$ where $\delta_{ps}^{(i)} = (p, l_0^{(i)}, b_0^{(i)}, l_1^{(i)}, b_1^{(i)}, ..., l^{(i)}_{|\delta_{ps}^{(i)}|-1}, b^{(i)}_{|\delta_{ps}^{(i)}|-1}, l^{(i)}_{|\delta_{ps}^{(i)}|}, s)$

Note that $b_0^{(i)}$ refers to the same broker for all $0 \leq i < m$, which is publisher p's home broker. Likewise, $b^{(i)}_{|\delta_{ps}^{(i)}|-1}$ refers to the same broker, which is the subscriber s's home broker. Let b_0 and $b_{|\delta_{ps}|-1}$ denote publisher p's home broker and subscriber s's home broker respectively. Hence, the pairwise reliability r'_{ps} between publisher p and subscriber s in dynamic tree scheme is estimated as

$$\begin{aligned} r'_{ps} &= \text{P}[\mathbb{G} \text{ is connected between } p \text{ and } s] \\ &\geq \text{P}[\mathbb{G}' \text{ is connected between } p \text{ and } s] \\ &\geq \text{P}[p\text{'s home broker is on}] \cdot \\ &\quad\ \text{P}[s\text{'s home broker is on}] \cdot \\ &\quad\ \text{P}[\text{at least one path is connected}] \\ &\geq a_{b_0} \cdot a_{b_{|\delta_{ps}|-1}} \cdot \\ &\quad\ (1 - \Pi_{i=0}^{m-1}(1 - \frac{r_{ps}^{(i)}}{a_{b_0} \cdot a_{b_{|\delta_{ps}|-1}}})) \end{aligned} \tag{12}$$

where $r_{ps}^{(i)}$ is the pairwise reliability of each path $\delta_{ps}^{(i)}$ in subgraph $\mathbb{G}'$, which can be calculated by Equation (3).

4.5 Path Bypassing + Event Buffering

We can combine the path bypassing scheme with the event buffering scheme in order to exploit path diversity as well as guarantee eventual delivery as follows. Each broker uses the event acknowledgement / buffering scheme as mentioned in Section 4.3. When a broker b_1 detects its neighbor b_2's failure, it uses the bypass routing *without* acknowledgement to forward the event to the failed broker b_2's immediately reachable children. The broker b_1 also keeps the event in its buffer and keeps retransmitting the event to the failed broker b_2 until b_2 recovers, receives the event, and sends the acknowledgement back to b_1. b_1 then discards the event. This scheme combines eventual delivery guarantee of the retransmission scheme with path diversity of the path bypassing scheme. The drawback of this approach is the additional overhead and potential event duplication at the subscribers. Event duplication, however, can be filtered out at the last-hop broker.

We can calculate the publisher-subscriber pairwise reliability for the path bypassing with event buffering scheme as follows. Let r'^{A}_{ps} be the estimated publisher-subscriber pairwise reliability for the path bypassing scheme (i.e., Equation (12) and d_{ps} be the end-to-end buffering delay for the event buffering scheme (i.e., Equation (11)). We can calculate the estimated pairwise reliability for the combined scheme as

$$\begin{aligned} r'_{ps} &= \text{P[event delivered immediately]} + \\ &\quad\ \text{P[partition]} \cdot \text{P[event delivered on time]} \\ &= r'^{A}_{ps} + (1 - r'^{A}_{ps}) \cdot \frac{\text{P}[d_{ps} \leq D]}{\text{P}[d_{ps} > 0]} \end{aligned} \tag{13}$$

where D is event lifetime. That is the total reliability is the probability that either the event can be delivered immediately via path bypassing scheme, or the event suffers network partition but the delay caused by the buffering scheme is still less than the event lifetime.

Note that estimated pairwise reliability r'_{ps} can be either calculated by Equation (3), Equation (4), Equation (12), or Equation (13), depending on which fault-tolerant technique is used. Once the estimated reliability r'_{ps} values of all publisher-subscriber pairs are calculated, they can be used to calculate the estimated subscriber reliability r'_s using Equation (2).

5 Evaluation

We evaluate the proposed analytical model via simulations with NS-2 network simulator [2]. In the simulator, we implement a topic-based, tree-based publish / subscribe protocol with four different reliability mechanisms described in Section 2.4 and analyzed in Section 4. In the experiment, we assume each publisher publishes its own unique topic. We assign a topic to each subscriber such that each topic's popularity follows Power law model.

5.1 Parameter Settings

We investigated several host availability traces and reports, ranging from commercial server log to distributed testbed log [3, 4, 18, 35]. In most cases, server's time between failures tends to range from several days to weeks, while time to repair usually range within hours.

Motivated by such finding, we describe a component[5] from availability perspective by two metrics, *period* and *availability*. We define the term *period* of a component as the summation of the component's mean time between failure and mean time to repair (i.e., mean failure-repair cycle length) and the term *availability* as the fraction of time the component is on. Thus, given a component x's period PR_x and availability a_x, we can calculate x's mean time between failures $MTBF_x = a_x.PR_x$ and mean time to repair $MTTR_x = (1 - a_x).PR_x$ respectively.

In the simulation, a component x will be *on* for the time period which is drawn from the exponential distribution with mean $MTBF_x$ before going to *off* state. Likewise, the component x will then be *off* for the time period drawn from the exponential distribution with mean $MTTR_x$ before going to *on* state again. Thus, such component x will have exponential failure rate $\gamma_x = \frac{1}{MTBF_x}$ and exponential repair rate $\sigma_x = \frac{1}{MTTR_x}$.

Based on the previous work [11], we set each overlay link's availability set to 0.99 and period to 60,000 seconds. Each publisher has default publishing interval equal to 1 minute. Each event has default lifetime equal to 3,600 seconds (1 hours). Each simulation is run for 14 days of simulation time. The evaluation result of each simulation parameter set is averaged from 10 runs.

5.2 Evaluation Results

We conduct the experiment with two sets of broker network topology. The first set is tree-based broker topology in order to study the performance of static path schemes (Section 4.2 and 4.3) without the effect of path diversity. The second set is random broker topology in order to study both static path schemes and path bypassing schemes (Section 4.4 and 4.5) with the effect of path diversity.

5.2.1 Tree-based Broker Network Topology

To study the performance of static-path reliability schemes (Section 4.2 and 4.3), we generate a random broker tree consisting of 10 brokers, 10 publishers, and 500 subscribers. We randomly assign a home broker to each publisher and each subscriber. We divide all generated trees into four sets. The first set of trees has each broker availability falling into [0.9, 0.95] range and use periodic subscription scheme. The second set of trees has each broker availability falling into [0.9, 0.95] range and use event buffering scheme. The third set of trees has each broker availability falling into [0.99, 0.999] range and use periodic subscription scheme. The fourth set of trees has each broker availability falling into [0.99, 0.999] range and use event buffering scheme. The [0.9, 0.95] availability range represents standard, off-the-shelf servers with low-to-moderate maintenance [3]. The [0.99, 0.999] availability range represents high-end, commercial servers with high maintenance [4].

[5]A component means a link or a broker

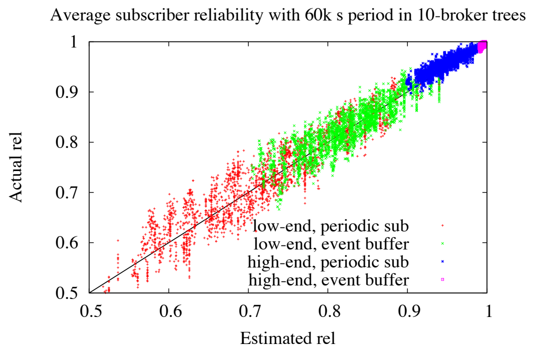
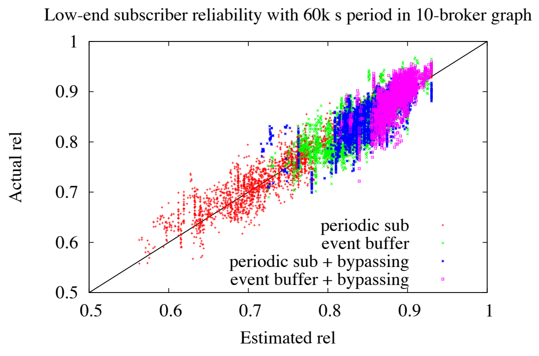


Figure 6: Subscriber reliability in static random tree with 10 brokers and 60K s period

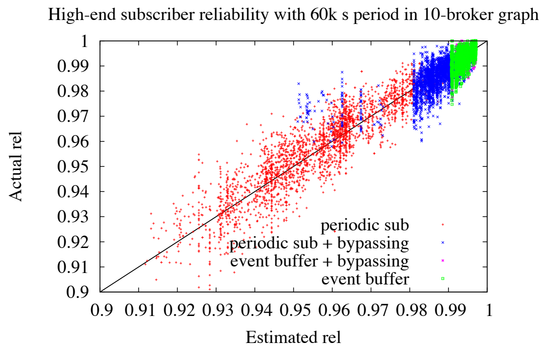
Figure 6 shows the predicted subscriber reliability on x axis and the actual subscriber reliability on y axis. Each point in the graph represents one subscriber. The color of the point represents the broker configuration and fault-tolerant scheme used. As shown in the graph, our proposed analytical model can accurately predict the subscriber reliability for each subscriber (i.e., all the points are clustered around x=y diagonal line). Also, there is a clear distinction of reliability value between different groups of broker configuration. The group with lowest reliability is the low-end servers with periodic subscription scheme, followed by the low-end servers with event buffering scheme. Notice that the event buffering scheme could achieve high reliability than the periodic subscription scheme, although the performance gain effect may be less, compared to the performance gain from the server's quality.

5.2.2 Random Broker Network Topology

We generate a random graph consisting of 10 brokers, 10 publishers, and 500 subscribers. Publishers and subscribers are randomly assigned to each broker. Again, we run the simulations with two broker availability specifications named *low-end* (0.9 - 0.95 availability) and *high-end* (0.99 - 0.999 availability). We compare the perfor-



(a) Low-end brokers



(b) High-end brokers

Figure 7: Subscriber reliability in 10-broker overlay graph with average degree 4 and 60K s period

mance in terms of subscriber reliability among four fault-tolerant techniques (i.e., from Section 4.2 - 4.5).

Figure 7 shows the performance comparison between the four protocols in 10-broker overlay graph. Again, our analytical model accurately predicts the real-time reliability of each subscriber. For the low-end broker configuration, the path bypassing scheme with event buffering has the best performance, followed by the path bypassing scheme with periodic subscription, the static path scheme with event buffering, and the static path scheme with periodic subscription respectively. However, for the high-end broker configuration, the static path scheme with event buffering performs best and as well as the path bypassing scheme with event buffering. This finding suggests that one should prefer to use the static path scheme with event buffering in high-end broker configuration, as it has lower overhead than the path bypassing scheme with event buffering.

6 Related Work

Improving reliability, timeliness, and other QoS metrics in wide-area overlay networks have been a significant topic of researchs for many years [5, 6, 12]. However, the approaches in this category are designed for point-to-point routing and do not specifically address decoupling nature between publishers and subscribers in publish/subscribe systems.

Several fault-tolerant mechanisms have been proposed specially for publish/subscribe systems under failures without considering timeliness property [8, 10, 13, 23]. On the other hand, several works have proposed performance analytical model to predict timeliness in publish/subscribe systems in perfect scenarios (i.e., without broker/link failures) [24,31,32]. This paper bridges such two approaches by quantitatively analyzing commonly used fault-tolerant techniques and their effect to *both* reliability and timeliness of publish/subscribe systems. There have been a few works that discuss timeliness of event delivery in publish/subscribe systems under component failures [14, 20]. However, they did not provide analytical model of their proposed systems. Recently, Esposito et al proposed and analyzed the use of network coding and gossiping to provide reliability and timeliness in tree-based publish / subscribe systems [15]. In contrast, our model discusses more commonly used fault-tolerant techniques such as path diversity and buffering. Also, our work does not assume tree-based publish / subscribe, but also works for any generic broker topology.

7 Conclusions

In this paper, we proposed an analytical model to estimate the subscriber real-time reliability for publish / subscribe systems with faulty brokers and links. We first described broker failure and link failure model before discussing several existing fault-tolerant techniques for distributed publish / subscribe systems. We then proposed the generic analytical model to estimate subscriber reliability. The evaluation via simulation has proved the correctness of our predictive model. Our proposed model can then be used as a building block for optimization problems such as subscriber assignment problem or broker network planning problem.

There are a few possible directions for future works. The first direction is to use the proposed analytical model to optimize performance of the publish / subscribe systems. Second, the analytical model could be extended to the case where component failure time is not exponentially distributed. Finally, another possible direction is to validate the proposed analytical model using data collected from real systems.

References

[1] Apache logging services. http://logging.apache.org/.

[2] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[3] Planetlab - all pairs pings. http://pdos.csail.mit.edu/ strib/pl_app/.

[4] Site5 uptime reports for all servers. http://www.site5.com/support/uptime/.

[5] Amir, Y., and Danilov, C. Reliable communication in overlay networks. In *DSN* (2003), IEEE Computer Society, pp. 511–520.

[6] Andersen, D. G. *Improving End-to-End Availability Using Overlay Networks*. Ph.D., Massachusetts Institute of Technology, Feb. 2005.

[7] Arianfar, S. Optimizing publish/subscribe systems with congestion handling. Master's thesis, Helsinki University of Technology, 2008.

[8] Chand, R., and Felber, P. Xnet: A reliable content-based publish/subscribe system. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 264–273.

[9] Cugola, G., and Jacobsen, H.-A. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 25–33.

[10] Cugola, G., Picco, G. P., and Murphy, A. L. Towards dynamic reconfiguration of distributed publish-subscribe middleware. In *SEM* (2002), A. Coen-Porisini and A. van der Hoek, Eds., vol. 2596 of *Lecture Notes in Computer Science*, Springer, pp. 187–202.

[11] Dahlin, M., Chandra, B. B. V., Gao, L., and Nayate, A. End-to-end wan service availability. *IEEE/ACM Trans. Netw. 11*, 2 (2003), 300–313.

[12] Duan, Z., Zhang, Z.-L., and Hou, Y. T. Service overlay networks: Slas, qos, and bandwidth provisioning. *IEEE/ACM Trans. Netw. 11*, 6 (2003), 870–883.

[13] Esposito, C., Cotroneo, D., and Gokhale, A. S. Reliable publish/subscribe middleware for time-sensitive internet-scale applications. In *DEBS* (2009), A. S. Gokhale and D. C. Schmidt, Eds., ACM.

[14] Esposito, C., Cotroneo, D., and Russo, S. Reliable event dissemination over wide-area networks without severe performance fluctuations. In *ISORC '10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (Washington, DC, USA, 2010), IEEE Computer Society, pp. 97–101.

[15] Esposito, C., Russo, S., Beraldi, R., Platania, M., and Baldoni, R. Achieving reliable and timely event dissemination over wan. In *Proceedings of the 13th international conference on Distributed Computing and Networking* (Berlin, Heidelberg, 2012), ICDCN'12, Springer-Verlag, pp. 265–280.

[16] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (2003), 114–131.

[17] Gerhards, R. The Syslog Protocol. RFC 5424 (Proposed Standard), March 2009.

[18] Godfrey, P. B. Repository of availability traces. http://www.cs.uiuc.edu/homes/pbg/availability/.

[19] Jaeger, M. A. *Self-Managing Publish/Subscribe Systems*. PhD thesis, Technische Universität Berlin, 2007.

[20] Jerzak, Z., and Fetzer, C. Soft state in publish/subscribe. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2009), ACM, pp. 1–12.

[21] Kalbfleisch, J. D., and Prentice, R. L. *The statistical analysis of failure time data*. Wiley-Interscience, 2011.

[22] Kazemzadeh, R. S., and Jacobsen, H.-A. Reliable and highly available distributed publish/subscribe service. In *SRDS '09: Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 41–50.

[23] Kazemzadeh, R. S., and Jacobsen, H.-A. Opportunistic multipath forwarding in content-based publish/subscribe overlays. In *Middleware* (2012), pp. 249–270.

[24] Kounev, S., Sachs, K., Bacon, J., and Buchmann, A. P. A methodology for performance modeling of distributed event-based systems. In *ISORC* (2008), pp. 13–22.

[25] Latouche, G., and Ramaswami, V. *Introduction to Matrix Analytic Methods in Stochastic Modelling*, 1 ed. ASA SIAM, 1999, ch. 2: PH Distributions.

[26] Li, M., Ye, F., Kim, M., Chen, H., and Lei, H. A scalable and elastic publish/subscribe service. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium* (Washington, DC, USA, 2011), IPDPS '11, IEEE Computer Society, pp. 1254–1265.

[27] Moler, C., and Loan, C. V. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review* (1978), 801–836.

[28] Muhl, G. *Large-scale Content-based Publish/Subscribe Systems*. PhD thesis, University of Technology Darmstadt, 2002.

[29] Pongthawornkamol, T., and Nahrstedt, K. Towards timeliness and reliability analysis of distributed content-based publish/subscribe systems over best-effort networks. Tech. Rep. http://hdl.handle.net/2142/14415, University of Illinois at Urbana-Champaign, November 2009.

[30] Pongthawornkamol, T., Nahrstedt, K., and Wang, G. Probabilistic qos modeling for reliability/timeliness prediction in distributed content-based publish/subscribe systems over best-effort networks. In *ICAC '10: Proceeding of the 7th international conference on Autonomic computing* (New York, NY, USA, 2010), ACM, pp. 185–194.

[31] Sachs, K. *Performance Modeling and Benchmarking of Event-Based Systems*. PhD thesis, TU Darmstadt, 2010. SPEC Distinguished Dissertation Award 2011.

[32] Schröter, A., Mühl, G., Kounev, S., Parzyjegla, H., and Richling, J. Stochastic performance analysis and capacity planning of publish/subscribe systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2010), DEBS '10, ACM, pp. 258–269.

[33] Shankar, A. U., and Lam, S. S. Time-dependent distributed systems: Proving safety, liveness and real-timeproperties. Tech. rep., Austin, TX, USA, 1985.

[34] Valiant, L. G. The complexity of enumeration and reliability problems. *SIAM Journal on Computing 8*, 3 (1979), 410–421.

[35] Yalagandula, P., Nath, S., Yu, H., Gibbons, P. B., and Seshan, S. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *In Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS* (2004).

[36] Zhang, C., Krishnamurthy, A., Wang, R. Y., and Singh, J. P. Combining flexibility and scalability in a peer-to-peer publish/subscribe system. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware* (New York, NY, USA, 2005), Middleware '05, Springer-Verlag New York, Inc., pp. 102–123.

Mitigating Anonymity Challenges in Automated Testing and Debugging Systems

Silviu Andrica and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Modern software often provides automated testing and bug reporting facilities that enable developers to improve the software after release. Alas, this comes at the cost of user anonymity: reported execution traces may identify users. We present a way to mitigate this inherent tension between developer utility and user anonymity: automatically transform execution traces in a way that preserves their utility for testing and debugging while, at the same time, providing *k*-anonymity to users, i.e., a guarantee that the trace can at most identify the user as being part of a group of *k* indistinguishable users. We evaluate this approach in the context of an automated testing and bug reporting system for smartphone applications.

1 Introduction

To debug a software failure, one must understand its root cause; unfortunately this can be quite challenging, with many bugs taking weeks to diagnose [1]. Therefore, modern software often ships with built-in features for automatically collecting program execution information that enables developers to more quickly debug the software (e.g., Windows Error Reporting collected billions of traces that helped developers fix 5,000 bugs [2]).

Current error reporting systems sacrifice developer productivity to preserve user anonymity: they report some execution information (e.g., backtraces, some memory contents) but forgo other useful information, such as the data the program was processing and the execution path it was following when it crashed, due to user privacy and anonymity concerns. In this paper, we seek to strike a better balance between user anonymity and productivity-enhancing execution information.

We describe this technique in the context of ReMoTe, an automated testing and debugging system that helps programs to collaborate on doing some of the debugging work that developers do: ReMoTe records program executions, modifies them to generate tests, runs the tests, and validates discovered bugs. For each part of this process, ReMoTe provides a so-called *pod* (Figure 1); pods from different program instances collaborate via a *hive*.

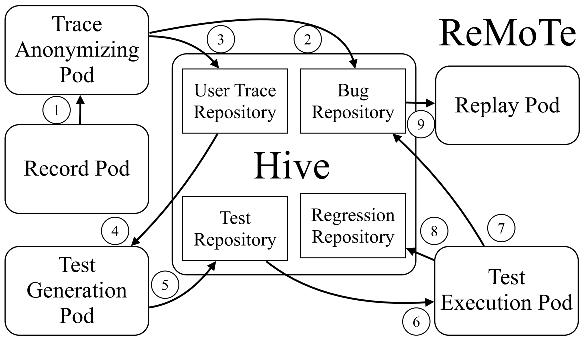


Figure 1: Overview of ReMoTe's architecture.

The *Record* pod logs a program's execution to an *interaction trace* that contains the program's interaction with the user and with the program's environment. Traces are added to local storage, but never removed.

ReMoTe is a distributed system: pods running on different machines can collaborate to generate tests, execute tests, and validate uncovered bugs by exchanging interaction traces. However, these traces contain information that may identify users. Thus, before a trace is shared with other pods, the *Trace Anonymizing* pod automatically transforms it to preserve the user's anonymity ①. Next, the pod informs the user about the amount of information the transformed trace contains that can identify her, and enables the user to veto the sharing.

There are two reasons why a ReMoTe user shares a trace with other pods: either the user experienced a bug, and the trace is put in the *Bug Repository* ②, or the user wishes to contribute the trace as a usage scenario, and the trace is put in the *User Trace Repository* ③. Developers can inspect the *Bug Repository* and use the *Replay* pod to replay bugs and understand how they occur ⑨.

The *Test Generation* pods modify traces from the *User Trace Repository* ④ to generate tests that populate the *Tests Repository* ⑤. *Test Execution* pods run these tests ⑥ and categorize them into bugs ⑦ and/or tests for the regression suite ⑧. These pods run inside the program that generated a trace or inside other program instances.

This paper presents the *Trace Anonymizing* pod. We describe means to quantify the degree of user anonymity and the utility of a trace, and devise algorithms to improve anonymity without harming utility.

First, we quantify the anonymity of a user sharing a trace as the size k of the set of distinct users reporting an equivalent trace. Second, we define a trace to have debugging utility if replaying it triggers the root cause of the bug and the failure [3], and to have test-generation utility if it describes an execution generated by an actual user. Third, our system improves user anonymity by expunging personally identifiable information and ensuring the user behavior encoded in a trace is not unique to that user, while still maintaining replayability of the trace.

The contributions of this paper are: 1) two techniques that provide users with k-anonymity, one using dynamic program analysis, and another leveraging crowdsourcing; and 2) a new metric that quantifies the amount of personally identifiable information contained in a trace. We built a ReMoTe prototype for Android applications and showed that ReMoTe protects users' anonymity ($k > 100$) and is more efficient than similar techniques.

In the rest of the paper, we define k-anonymity (§2), describe the two algorithms (§3–§4), evaluate our prototype (§5), review related work (§6), and conclude (§7).

2 Anonymity of Interaction Traces

This section defines interaction traces, describes the concept of k-anonymity that underlies our work, and defines a metric to quantify the amount of user-identifying information contained in a trace.

2.1 Interaction Traces and Event Types

We define an interaction trace T as a sequence of events, $T =< e_1, \ldots, e_n >$. Each event e_i records one of four sources of "non-determinism" that influence a program's execution: 1) user interaction with the program's GUI, 2) network communication, 3) input read from the machine the program runs on, and 4) decisions made by the runtime system. Replaying a trace T should consistently drive the program along the same execution path.

An event plays one of two roles during replay: *proactive events* cause a feature of the program to execute (e.g., click on the "Send SMS" button), while *reactive events* provide the feature with the input it needs (e.g., the phone number and SMS text). Events of both types may contain information that identifies users. Table 1 shows events for each source of non-determinism for interactive Android applications, and maps them to a role.

ReMoTe targets interactive programs, which generate these events at a rate bounded by the speed with which users interact with programs. Thus, compared to recording solutions that target events at a lower software layer (e.g., [4]), which are generated more frequently, the *Record* pod is more scalable, since it runs less frequently.

	User	**Network**	**Device**	**Runtime**
Proactive	Tap, Tilt, Scroll, Drag, Press key	Receive push notification	Trigger geofence	Fire timer
Reactive	Textbox value, Selector value, Slider value	Server response	Date, GPS location, Camera & Mic output, Shared data, Device settings	Async tasks scheduling

Table 1: Trace events for Android applications, classified by covered non-determinism source and proactivity role.

We say a trace contains personally identifiable information (PII) if it can be used to determine a user's identity, either alone or when combined with other information that is linkable to a specific user [5].

2.2 K-Anonymity

A data set satisfies k-anonymity if and only if each set of values that can be used to identify the source of a data element appears at least k times in the set [6], i.e., the source of an element cannot be narrowed down to fewer than k candidates. We say that each element of a data set satisfying k-anonymity is k-anonymous.

In [6], the data set is represented as a table PT, and each row contains information about a single subject. Some table columns contain private information (e.g., received medication), others provide identification details about the subject (e.g., birth date and zip code), but none contain information that explicitly identifies the subject (e.g., the name of the patient). Thus, one may naïvely conclude that table PT is anonymous.

K-anonymity quantifies the possibility of linking entries from the PT table with external information to infer the identities of the sources of the data in the PT table.

Consider there exists a set QI of columns in PT, called a quasi-identifier, (e.g., $QI = \{birth\ date, zipcode\}, PT = QI \bigcup \{medication\}$) that also appears in a publicly available table PAT. If the PAT table contains additional columns that explicitly identify its sources (e.g., $PAT = Voters\ list = \{name\} \bigcup QI$), then an attacker can use the quasi-identifier values to join the two tables and learn private information about a subject (e.g., the medication a person receives). The attack is similar to executing an SQL join operation on the PT and PAT tables that uses the quasi-identifier as the join condition.

This attack relies on the value of the quasi-identifier being unique for each subject in the *PT* and *PAT* tables. To achieve k-anonymity, one must modify the *PT* table to break this assumption [7]. This is not necessary if, in the *PT* table, each quasi-identifier value already appears k times. If not, one can suppress the entries that prevent achieving k-anonymity, or repeatedly use generalization strategies to make the values of the quasi-identifier less precise (e.g., replace the birth date with the year of birth) until k-anonymity is reached, or add new entries to the table to make it satisfy k-anonymity (not covered in [7]).

We seek to prevent ill-intentioned developers and program users from abusing interaction traces to learn the identity of the user whose program recorded a trace.

A trace identifies its source through reactive events, which may contain explicit PII (e.g., usernames), or through proactive events, which detail user behavior. We aim to provide users with k-anonymity, which in our case represents the guarantee that a trace identifies its source as the member of a set of k indistinguishable users.

We say an interaction trace is k-anonymous if it is k-proactive-anonymous and k-reactive-anonymous. A trace is k-reactive-anonymous if, for each reactive event in the trace, there exist at least k alternatives (§3). A trace is k-proactive-anonymous if at least k users observed it (§4). Thus, a k-anonymous trace contains behavior exhibited by k users, and there are k alternatives for each program input contained in a reactive event.

We now describe the differences between the original k-anonymity technique [6] and ours:

First, ReMoTe computes the maximal k it can achieve for a trace's anonymity, it does not enforce a particular k.

Second, ReMoTe cannot detect a minimum, complete quasi-identifier, as is assumed in [6], because the structure of a trace is unconstrained and its length is unbounded. ReMoTe takes a conservative approach, by choosing completeness over minimality, and defines the quasi-identifier to span all the events in a trace.

Third, the equivalent of the *PT* table is distributed among users. While the ReMoTe hive could store all the observed, non-anonymized traces, doing so posses the risk of an attacker subverting the hive, gaining access to the raw traces, and thus being able to identify users.

Finally, the pods share a trace with the hive only once it has achieved k-anonymity. K-anonymity increases, for example, when adding a newly recorded trace to the set causes existing ones to become k-proactive-anonymous.

2.3 Amount of Disclosed Information

We define the *k-disclosure* metric to quantify the amount of PII in a trace T. We start from two observations: First, its value should be inversely proportional to how k-anonymous an observed trace T is, because the higher the k, the less specific to a user the trace is. Second, the amount of PII contained in a trace is emergent: while each event in the trace may be encountered by multiple users, the order of events in the trace may be unique.

We define the value of the k-disclosure metric for an observed trace T, $k\text{-disclosure}(T)$, to be the sum of the inverses of the values quantifying how k-anonymous is each of T's subsequences, $k(trace)$. That is, $k\text{-disclosure}(T) = \sum_{1 \leq i \leq j \leq |T|} \frac{1}{k(T_{ij} = <e_i, \ldots, e_j>)}$.

We expect $k\text{-disclosure}(T)$ to decrease over time because, once a program observes a trace, it is permanently added to local storage; as more users encounter T or its subsequences, the trace's k-anonymity increases.

3 Anonymity of Reactive Events

Reactive events contain program inputs that can directly identify users, such as usernames. A reactive event is useful for replaying a trace T if it causes the program to make the same decisions during replay as it did during recording [3]. If one can replace a reactive event e_R^{orig} with $k-1$ reactive events e_R^{sub} without affecting the trace's ability to replay the program execution, then we say the trace T is k-reactive-anonymous with respect to event e_R^{orig}. More simply, we say e_R^{orig} is k-anonymous.

Consider the example of an Android application for sending SMS messages. The user fills in the destination phone number (reactive event e_R) and the message body. When the user presses the "Send" button, the application converts the phone number to a `long`. Say that the user entered a number starting with a `'+'` character, and the program crashes, but any string that does not start with a digit can reproduce this crash—ReMoTe can replace e_R with k alternatives, where k is the number of such strings.

To compute the number k of alternatives for a reactive event e_R, ReMoTe must know how the program makes decisions based on the program input associated with e_R. ReMoTe uses concolic execution [8] to collect the conditions, called path constraints, corresponding to the executed branch statements that depend on reactive events. The technique uses "symbolic" variables that encode constraints on values, instead of concrete values.

Next, for each reactive event, ReMoTe uses a constraint solver to compute the solutions that satisfy the path constraints referring to it. The number of solutions determines how anonymous trace T is w.r.t. that event.

ReMoTe uses the following algorithm. It replays each event e_i in a trace T. When replaying a reactive event, the algorithm copies $e_i.input$ (the program input contained in e_i) to $e_i.concrete$, marks $e_i.input$ symbolic, and adds an entry for e_i in the map tracking path constraints (*PC*). When the program branches on a condition involving the symbolic variable $e_j.input$, and both branch targets may be followed, the algorithm forces the program to take the target that e_j*.concrete* satisfies. The algorithm uses static analysis to decide whether to add the path constraint corresponding to the taken branch to the *PC* map and maintain e_j*.input* symbolic, or to replace it with e_j*.concrete*. When replay finishes, the algorithm computes the number of solutions for each reactive event e_R.

ReMoTe iteratively computes the number of solutions for a set of path constraints *PC* by generating a solution, adding its negation to *PC*, and asking the solver for another solution. This process is time consuming, so ReMoTe bounds the number of solutions, establishing a lower bound for how *k*-reactive-anonymous is a trace.

ReMoTe modifies the trace to replace each program input contained in a reactive event with one of its computed alternatives, thus removing the PII from the trace.

The algorithm is similar to the one described in [9]. The difference is our use of static analysis to make concolic execution more efficient by avoiding the concolic execution of runtime-system code. This code affects only the execution of the runtime system, not the execution of the program and, thus, needlessly slows down concolic execution. The static analysis examines the stack trace of a program when it branches on a symbolic variable, and checks if the branch is *in* the program's code or if its result is *used* by the program—only in these two cases is the associated path constraint added to the *PC* map.

A technical report [10] presents a detailed description of the algorithm, the benefits of using the generated alternatives, their drawbacks, and mitigation solutions.

4 Anonymity of Proactive Events

Proactive events reveal a program's usage, and this usage could uniquely identify the user. For example, an employee may access a company application's features that are only accessible to executive management, and then a feature only accessible to financial department employees. By analyzing the corresponding proactive events, one could infer that the user is the company's CFO.

A related example is one where the two features are accessed in different traces, and each trace contains the same sequence of events that acts as a quasi-identifier and allows identifying the user.

To prevent user behavior details contained in a trace from identifying users, ReMoTe ensures that the entire trace (including reactive events) is *k*-proactive-anonymous *before* it is passed to the hive. Therefore, every trace seen by the hive corresponds to the executions of $\geq k$ distinct users, and it does not contain behavior specific to any one of them, so the trace's source cannot be identified more narrowly than that set of $\geq k$ users.

To check if a trace T is *k*-proactive-anonymous, ReMoTe can just query every program instance whether it experienced execution trace T in the past, and tally up the results. Alas, providing trace T to other program instances could compromise the user's anonymity.

The challenge is to design an algorithm that counts how many users observed the trace T without explicitly revealing T to them. Our solution is to hide T among a set S of traces, ask program instances whether they observed any of the traces in S, and probabilistically compute the number k of instances that indeed observed T.

The algorithm runs as follows: Program instance A, run by user U who wishes to share the trace T, constructs the query set S. The set S contains the hashes of trace T and of other traces that act as noise. Next, instance A sends the set S to the ReMoTe hive, which forwards the set S to each program instance A_i run by users U_i.

After the *Record* pod records an interaction trace, it saves the hashes of the trace and of its sub-traces to a history set H. When receiving a query set S, each instance A_i replies positively if its history set H_i contains any of the hashes in the set S, i.e, if $H_i \bigcap S \neq \emptyset$.

The ReMoTe hive counts the number K of positive replies and sends it to instance A, which computes the probability that k of the K instances recorded T—this determines how *k*-proactive-anonymous trace T is.

This algorithm protects the anonymity of the U and U_i users, because instances A_i cannot learn T, and instance A cannot learn the trace that caused A_i to reply positively.

ReMoTe runs the same algorithm for each of trace T's sub-traces and computes the amount of personally identifiable information contained in trace T, i.e., *k*-disclosure(T). Finally, instance A reports the k and *k*-disclosure(T) values to the user U. If the user agrees, instance A shares the trace T with the ReMoTe hive.

There are four challenges associated with this algorithm. First, instance A may be tricked into revealing the trace T by a sequence of carefully crafted queries. Second, instance A needs to generate feasible traces as noise for the set S. Third, to compute the number k of instances that recorded T, instance A must approximate the likelihood of instances A_i recording each trace from the set S. Finally, instance A may be tricked into revealing T by an attacker compromising the result of the voting process. [10] details these challenges and presents their solutions.

5 Empirical Evaluation

We built a ReMoTe prototype that can be used by Android applications. In this section, we evaluate it on PocketCampus [11], a client-server Android application that we modified to use ReMoTe.

We evaluate the anonymity ReMoTe can provide for reactive events by quantifying how *k*-anonymous each field of a server response is after PocketCampus processes it. We focus on two functionalities provided by the application: check the balance of a student card account, and display the time of the next departure from a public transport station. In both cases, PocketCampus makes a request to a server, processes the response, and displays some information. Each server response contains multiple fields, and we computed how many alternatives ReMoTe can generate for each field.

Figure 2 contains the results, and shows that PocketCampus places few constraints on server responses, enabling ReMoTe to provide high k-anonymity for users.

Figure 2a shows that the server's response to inquires about the account balance contains seven fields: one is not processed by PocketCampus (the white box), three for which ReMoTe generates more than 100 alternatives (the gray boxes), and three for which ReMoTe cannot generate alternatives (the black boxes), because they contain floating point values (not supported by our constraint solver) or because PocketCampus checks their value against constant values (e.g., the server's status).

Figure 2b shows the server's response when queried about the name of a public transport station.

Figures 2c and 2d show the same server response, but in different interaction traces. Figure 2c corresponds to PocketCampus showing the departure time, while Figure 2d corresponds to additionally displaying trip details, which causes PocketCampus to further process the response and place additional constraints.

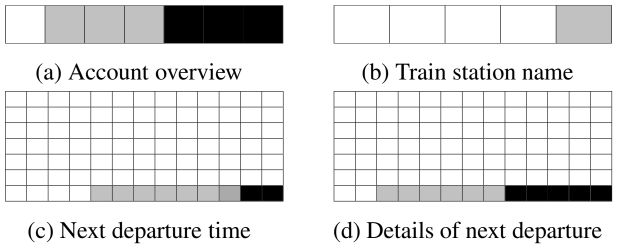


Figure 2: k-anonymity for server response fields. White boxes show unprocessed fields, gray boxes show fields with $k \geq 100$, black boxes show fields with $k = 1$.

We report the percentage of bits identical in the original server response fields and the ones ReMoTe generated, considering only those processed by PocketCampus. Figure 3 shows that the alternative responses reveal, on average, 72% of the original bits, thus being more similar to the original ones than randomly-generated ones.

We evaluate the speedup in the concolic execution completion time brought by using the static analysis described in Section 3. Figure 4 shows that, when using the analysis' results, PocketCampus finishes processing a server response within 10 minutes, as opposed to more than one hour when the analysis is not used.

The technical report [10] evaluates additional aspects of ReMoTe, such as how k-proactive-anonymous is a

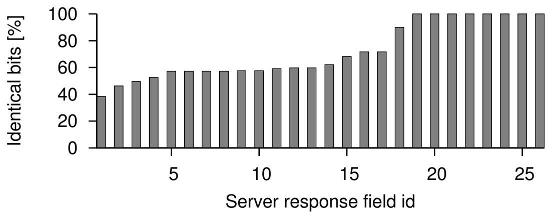


Figure 3: Percentage of bits identical in the original server responses and the alternatives ReMoTe generated.

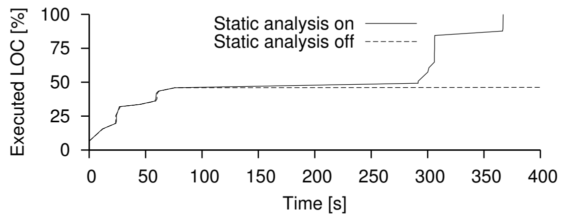


Figure 4: Concolic execution speedup obtained by using the static analysis described in Section 3. The figure can be read as a comparison, in terms of efficiency, between our algorithm and the one described in [9].

trace and what is its storage space overhead.

6 Related Work

Our work on generating alternatives for reactive events is most similar to [9], which also relies on collecting path constraints that describe decisions made by a program. The differences are that we use static analysis to discard path constraints that do not affect a program's execution, and we consider the anonymity threats related to a user interacting with a program. Camouflage [12] builds on [9] and introduces two techniques to enlarge the set of bug-triggering inputs, which ReMoTe can leverage.

Our crowdsourced k-anonymity technique is similar to the query restriction techniques pertaining to statistical databases [13], since one can view the crowd of program instances as a distributed database.

7 Conclusions

This paper describes two techniques to transform program execution traces to maximize users' anonymity, yet maintain the traces' utility for testing and debugging. One technique uses dynamic program analysis to expunge explicit personally identifiable information from traces, while the other leverages the crowd of users to verify that the user behavior encoded in a trace is not unique. We prototyped the techniques, and preliminary results suggest that they are effective and efficient.

References

[1] P. Godefroid and N. Nagappan, "Concurrency at Microsoft – An exploratory survey," in *Intl. Conf. on Computer Aided Verification*, 2008.

[2] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *Symp. on Operating Systems Principles*, 2009.

[3] C. Zamfir, G. Altekar, G. Candea, and I. Stoica, "Debug determinism: The sweet spot for replay-based debugging," in *Workshop on Hot Topics in Operating Systems*, 2011.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Symp. on Operating Sys. Design and Implem.*, 2002.

[5] B. Krishnamurthy and C. E. Wills, "On the leakage of personally identifiable information via online social networks," in *Workshop on Online Social Networks*, 2009.

[6] L. Sweeney, "K-Anonymity: A model for protecting privacy," in *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.

[7] P. Samarati and L. Sweeney, "Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression," SRI International, Tech. Rep., 1998.

[8] K. Sen, "Concolic testing," in *Intl. Conf. on Automated Software Engineering*, 2007.

[9] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[10] S. Andrica and G. Candea, "Mitigating anonymity challenges in automated testing and debugging systems," Ecole Polytechnique Fédérale de Lausanne (EPFL), Tech. Rep. #186656, 2013.

[11] "PocketCampus," http://www.pocketcampus.org.

[12] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Intl. Conf. on Software Engineering*, 2011.

[13] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *ACM SIGMOD Conf.*, 2000.

Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs

Christopher Stewart, Aniket Chakrabarti
The Ohio State University

Rean Griffith
VMWare

Abstract

Internet services access networked storage many times while processing a request. Just a few slow storage accesses per request can raise response times a lot, making the whole service less usable and hurting profits. This paper presents Zoolander, a key value store that meets strict, low latency service level objectives (SLOs). Zoolander scales out using replication for predictability, an old but seldom-used approach that uses redundant accesses to mask outlier response times. Zoolander also scales out using traditional replication and partitioning. It uses an analytic model to efficiently combine these competing approaches based on systems data and workload conditions. For example, when workloads under utilize system resources, Zoolander's model often suggests replication for predictability, strengthening service levels by reducing outlier response times. When workloads use system resources heavily, causing large queuing delays, Zoolander's model suggests scaling out via traditional approaches. We used a diurnal trace to test Zoolander at scale (up to 40M accesses per hour). Zoolander reduced SLO violations by 32%.

1 Introduction

Internet services built on top of networked storage expect data accesses to complete quickly all of the time. Many companies now include latency clauses in the service level objectives (SLOs) given to storage managers. Such SLOs may read, "98% of all storage accesses should complete within 300ms provided the arrival rate is below 500 accesses per second [12, 35, 39]." When these SLOs are violated, Internet services become less usable and earn less revenue. Consider e-commerce services. SLO violations delay web page loading times. As a rule of thumb, delays exceeding 100ms decrease total revenue by 1% [30]. Such delays are costly because revenue, which covers salaries, marketing, etc., far exceeds the cost of networked storage. A 1% drop in revenue can cost more than an 11% increase in compute costs [38].

Many networked storage systems meet their SLOs by scaling out, i.e., when access rates increase, they add new nodes. The most widely used scale-out approaches partition or replicate data from old nodes to new nodes and divide storage accesses across the old and new nodes, reducing resource contention and increasing throughput [12, 15, 24]. However, background jobs, e.g., write-buffer dumps, garbage collection, and DNS timeouts, also contend for resources. These periodic events can increase access times by several orders of magnitude.

Our key-value store, called Zoolander, masks slow storage accesses via replication for predictability, a historically dumb idea whose time has come [29]. Replication for predictability scales out by copying the exact same data across multiple nodes (each node is called a duplicate), sending all read/write accesses to each duplicate, and using the first result received. Historically, this approach has been dismissed because adding a duplicate does not increase throughput. But duplicates can reduce the chances for a storage access to be delayed by a background job, shrinking heavy tails[1] Very recent work has used replication for predictability but only sparingly with ad-hoc goals [2, 9, 39]. Zoolander fully supports replication for predictability at scale.

Zoolander can also scale out by reducing the accesses per node using partitioning and traditional replication. Its policy is to selectively use replication for predictability only when it is the most efficient way to scale out (i.e., it can meet SLO using fewer nodes than the traditional approaches). Zoolander implements this policy via a biased analytic model that predicts service levels for 1) the traditional approaches under ideal conditions and 2) replication for predictability under actual conditions. Specifically, the model assumes that accesses will be evenly divided across nodes (i.e., no hot spots). As a result, the model overestimates performance for traditional approaches. In contrast, our model predicts the performance of replication for predictability precisely, using first principles and measured systems data. Despite its bias, our model provided key insights. First, replication for predictability allows us to support very strict, low latency SLOs that traditional approaches cannot attain. Second, traditional approaches provide efficient scale out when system resources are heavily loaded, but replication for predictability can be the more efficient approach when resources are well provisioned.

We implemented Zoolander as a middleware for existing key-value stores, building on prior designs for high

[1]In this paper, we use the term *heavy tailed* to describe probability distributions that are skewed relative to normal distributions. Sometimes these distributions are called fat tailed.

throughput [16, 18, 39]. Zoolander extends these systems with the following features:

1. High throughput and strong SLO for read and write accesses when clients do not share keys. Zoolander also supports shared keys but with lower throughput.

2. Low latency along the shared path to duplicates via reduced TCP handshakes and client-side callbacks.

3. Reuse of existing replicas to reduce bandwidth needs.

4. A framework for fault tolerance and online adaptation.

We used write- and read-only benchmarks to validate Zoolander's analytic model for replication for predictability under scale out. The model predicted actual service levels, i.e., the percentage of access times within SLO latency bounds, within 0.03 percentage points.Replication for predictability increased service levels significantly. On the write-only workload using 4 nodes, Zoolander achieved access times within 15ms with a 4-nines service level (99.991%). Using the same number of nodes, traditional approaches achieved a service level of only 99%—Zoolander increased service levels by 2 orders of magnitude.

We set up Zoolander on 144 EC2 units and issued up to 40M accesses per hour, nearly matching access rates seen by popular e-commerce services [4, 7, 17]. We also varied the access rate in a diurnal pattern [34]. By using both replication for predictability and traditional approaches, Zoolander provided new, cost effective ways to scale. At night time, when arrival rates drop, Zoolander decided not to turn off under used nodes. Instead, it used them to reduce costly SLO violations. Zoolander's approach reduced nightly operating costs by 21%, given cost data from [17, 38]. With better data migration, Zoolander could have reduced costs by 32%.

This paper is arranged as follows: Section 2 presents Zoolander's analytic model on SLO under replication for predictability. Section 3 describes Zoolander itself and compares achieved SLOs to model predictions. Section 4 offers model-driven insights on when to use replication for predictability. Section 5 studies Zoolander at scale on EC2. Section 7 concludes.

2 Replication for Predictability

Traditional approaches to scale out networked storage share a common goal: They try to reduce accesses per node by adding nodes. While such approaches improve throughput, there is a downside. By sending each access to only 1 node, there is a chance that accessess will be delayed by background jobs on the node [9]. Normally, background jobs do not affect access times, but when they do interfere, they can cause large slowdowns. Consider write buffer flushing in Cassandra [16]. By default,

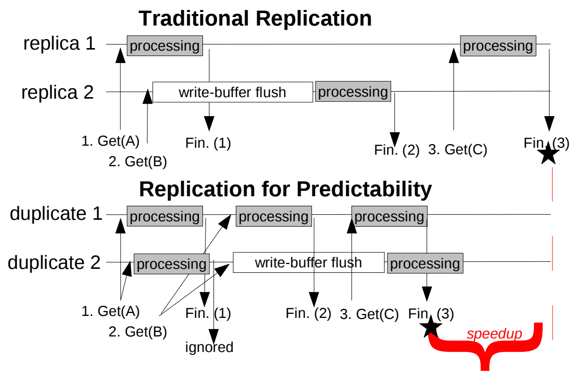


Figure 1: Replication for predictability versus traditional replication. Horizontal lines reflect each node's local time. Numbered commands reflect storage accesses. Get #3 depends on #1 and #2. Star reflects the client's perceived access time.

writes are committed to disk every 10 seconds by flushing an in-memory cache. The cache ensures that most writes proceed at full speed without incurring delay due to a disk access. However, if writes arrive randomly and buffer flushes take 50ms, we would expect buffer flushes to slow down 0.5% of write accesses ($\frac{50ms}{10s}$).

Figure 1 compares replication for predictability against traditional, divide-the-work replication. The latter processes each request on one node. When a buffer flush occurs, pending accesses must wait, possibly for a long time. However, by sending all accesses to N nodes and taking the result from the fastest, replication for predictability can mask $N-1$ slow accesses, albeit without scaling throughput. In this section, we generalize this example by modelling replication for predictability. Our analytic model outputs the expected number of storage accesses that complete within a latency bound. It allows us to compare replication for predictability to traditional approaches in terms of SLO achieved and cost.

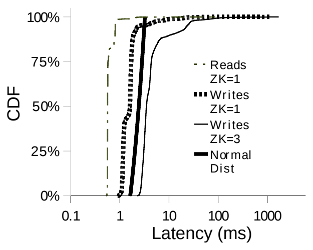
2.1 First Principles

Our model is based on the following first principles:

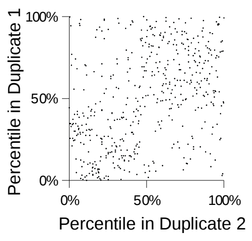
1. Outlier access times are heavy tailed. Background jobs can cause long delays, producing outliers that are slower and more frequent than Normal tails.

2. Outliers are non-deterministic with respect to duplicates. To mask outliers, slow accesses on 1 duplicate can not spread to others. Replication for predictability does not mask outliers caused by deterministic factors, e.g., hot spots, convoy effects, and poor workload locality.

To validate our first principles, we studied storage access times in our own local, private cloud. We use a 112 node cluster, where each node is a core with at least 2.4 GHz, 3MB L2 cache, 2GB of DRAM memory, and 100GB of secondary storage. Our virtualization software



(a)



(b)

Figure 2: Validation of our first principles. (A) Access times for Zookeeper under read- and write-only workloads exhibit heavy tails. (B) Outlier accesses on one duplicate are not always outliers on the other.

is User-Mode Linux (UML) [13], a port of the Linux operating system that runs in user space of any X86 Linux system. Thus, RedHat Linux (kernel 2.6.18) serves as our VMM. Custom PERL scripts designed in the mold of Usher [26] allow us to 1) run preset virtual machines on server hardware, 2) stop virtual machines, 3) create private networks, and 4) expose public IPs. Our cloud infrastructure is compatible with any public cloud that hosts X86 Linux instances. Later in this paper, we will scale out on Amazon EC2.

We set up Zookeeper [18] and performed $100,000$ data accesses one after another. Zookeeper is a key-value store that is widely used to synchronize distributed systems. It is deployed as a cluster with ZK nodes. Writes are seen by $\frac{ZK}{2}+1$ nodes. Reads are processed by only 1 node. Figure 2(a) plots the cumulative distribution function (CDF) for Zookeeper under read-only and write-only workloads. The coefficient of variation ($\frac{\sigma}{|\mu|}$), or COV, shows the normalized variation in a distribution. Generally, COV equal or below 1 is considered low variance. We compared the plots in Figure 2(a) by 1) computing COV before the tail, i.e., up to the 70^{th} percentile and 2) computing COV across the whole CDF. Before the tail, COV was below 1. Across the entire distribution, COV was much higher, ranging from 1.5–8.

To visually highlight the heaviness of the tails, Figure 2(a) also plots a normal distribution with standard deviation and mean that were 25% larger than 90% of write times in ZK=1. Note, COV in an normal distribution is 1. The tails for both reads and writes under ZK=1 overtake the normal distribution, even though the normal distribution has a larger mean. We also found that tails became heavier as complexity increased. Writes in a single-node Zookeeper led to local disk accesses that didn't happen under reads. Writes in 3-node Zookeeper groups send network messages for consistency.

We can also interpret each (x,y) point in Figure 2(a) as a latency bound and an achieved service level. If access times followed a normal distribution, a latency bound that was 3 times the mean would provide a service level of 99.8%. Figure 2(a) shows that Zookeeper's service levels were only 98.8% of reads,96.0% of 1-node writes, and 91.5% of 3-node writes under that latency bound. To support a strict SLO that could cover 99.99% of data accesses, the latency bound would have risen to 16X, 26X, and 99X relative to the means.

Heavy tails affect many key value stores, not just Zookeeper. Internal data from Google shows that a service level of 99.9% in a default, read-only BigTable setup would require a latency bound that is 31X larger than the mean [9]. Others have noticed similar results on production systems [6, 17]. We also measured read access times in a single Memcached node, a key-value widely used in practice and in emerging sustainable systems [4, 31]. We saw a coefficient of variation of 1.9, and, under a lax latency bound, only a 98.3% service level was achieved. Finally, we ran the same test with Cassandra [16], another widely used key-value store, deployed on large EC2 instances. The coefficient of variation was 6.4.

Figure 2(b) highlights principle #2. Across two Zookeeper runs that receive the same requests under no concurrency, we show the percentile of each storage access. If slow service times were workload dependent, either the bottom right or upper left quartiles of this plot would have been empty, i.e., slow accesses on the first run would be slow again on the second. Instead, every quartile was touched.

2.2 Analytic Model

This subsection references the symbols defined in Table 1. Our model characterizes the service level provided by N independent duplicates running the exact same workload. The latency bound (τ) for the SLO is given as input. Written in plain english, *our model predicts that $\hat{s}$ percent of requests will complete within τ ms.*

$\hat{s}$	Expected service level
N	Number of duplicates used to mask anomalies
τ	Target latency bound
$\Phi_n(k)$	Percentage of service times from duplicate n with latency below k
λ	Mean interarrival rate for storage accesses
μ_{net}	Mean of network latency between duplicates and storage clients
μ_{rep}	Mean delay to duplicate a message one time plus the delay to prune a tardy reply
μ_n	Mean service time for duplicate n (derived)

Table 1: Zoolander inputs.

Using principles #1 and 2, we first model the probability that the fastest duplicate will meet an SLO latency bound. Recall, writes are sent to all duplicates, so any duplicate can process any request. Handling failures is treated as an implementation issue, not a modelling issue. The probability that the fastest duplicate responds within latency bound is computed as follows:

$$\hat{s} = \sum_{n=0}^{N-1} [\Phi_n(\tau) * \prod_{i=0}^{n-1} (1 - \Phi_i(\tau))]$$

To provide intuition into this result, consider $\Phi_i(\tau)$ is the probability that duplicate i meets the τ ms latency bound. If $N = 2$, $\Phi_1(\tau) * (1 - \Phi_0(\tau))$ is the probability that duplicate 1 masks a SLO violation for duplicate 0. Intuitively, as we scale out in N, each term in the sum is the probability that the n^{th} duplicate is the firewall for meeting SLO, i.e., duplicates $0..(n-1)$ take too long to respond but n meets the bound. When all duplicates have the same service time distribution, we can reduce the above equation to a geometric series, shown below. (Note, as N approaches infinity, $\hat{s}$ converges to 1.)

$$\hat{s} = \sum_{n=0} \Phi_n(\tau) * (1 - \Phi_n(\tau))^n = 1 - (1 - \Phi_n(\tau))^N$$

Queuing and Network Delay: SLOs reflect a client's perceived latency which may include processing time, queuing delay, and network latency. Since duplicates execute the same workload, they share access arrival patterns and their respective queuing delays are correlated. Similarly, networking problems can affect all duplicates. Here, we lean on prior work on queuing theory to answer two questions. First, does the expected queuing level completely inhibit replication for predictability? And second, how many duplicates are needed to overcome the effects of queuing? The key idea is to deduct the queuing delay from τ in the base model. Intuitively, requiring all duplicates to reduce their expected service time in proportion to the expected queuing delay.

$$\tau_n = \tau - (\frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho} * \mu_n) - \mu_{net}$$

$$\hat{s} = \sum_{n=0}^{N-1} [\Phi_n(\tau_n) * \prod_{i=0}^{n-1} (1 - \Phi_i(\tau_i))]$$

We used an M/G/1 queuing model to derive the expected queuing delay, reflecting the heavy-tail service times observed in Figure 2(a). To briefly explain the first equation above, an M/G/1 models the expected queuing delay as a function of system utilization (ρ), distribution variance (C_v^2), and mean service time. Utilization is the mean arrival rate divided by the mean service time. Note, that the new τ may be different for each node (parameterizing it by n). An M/G/1 assumes that inter-arrivals are exponentially distributed. This may not be the case in all data-intensive services. A G/G/1 with some constraints on inter-arrival may be more accurate. Alternatively, an M/M/1 would have simplified our model, eliminating the need for the squared coefficient of variance (C_v^2). Prior work has shown that multi-class M/M/1 can sometimes capture the first-order effects of M/G/1.

We deduct the mean time lost to network latency. Here, network latency is the average delay to send a TCP message between any two nodes.

Multi-cast and Pruning Overhead: Replication for predictability incurs overhead when messages are repeated to all duplicates and when unused messages are pruned. These activities become more costly as the number of duplicates increase. We use a linear model to capture this. Note, we expect emerging routers to provide multi-cast support that reduces this overhead a lot. However, storage systems that use software multi-cast, like Zoolander, should consider this overhead.

$$\tau_n = \tau - (\frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho} * \mu_n) - \mu_{net} - N * \mu_{rep}$$

Discussion: With a nod toward systems builders, we kept the model simple and easy to understand. Most inputs come from CDF or arrival-rate data that can be collected using standard tools. The model does not capture non-linear correlations between outliers, resource dependencies, or the root causes of SLO violations.

3 Zoolander

Zoolander is middleware for existing key-value stores. It adds full read and write support for replication for predictability. Figure 3 highlights the key components of Zoolander. In the center of the figure, we show that keys are stored in *duplicates and partitions*. A duplicate abstracts an existing key-value store, e.g., Zookeeper or Cassandra. As such, a duplicate may span many nodes but it does not share resources with other duplicates.

A partition comprises 1 or more duplicates. Storage accesses are sent to all duplicates within a partition—i.e., duplicates implement replication for predictability. Storage accesses are sent to only 1 partition. There is no cross-partition communication. A global hash function maps keys to partitions. All of the keys mapped to a partition comprise a *shard*.

Zoolander can scale out by reducing storage accesses per node via partitioning. It can also scale out by adding duplicates. At the top of Figure 3, we highlight the Zoolander manager which uses our analytic model to scale out efficiently. The manager takes as input a target service level and latency bound. It also collects CDF data on service times, networking delays, and arrival rates per shard. The manager then uses our model from Section 2 to find a replication policy that meets the target SLO. It finds a policy by iteratively 1) moving a shard from one partition to another, 2) placing a shard on a new partition, and 3) adding/removing duplicates from a partition. The first and second options change the arrival rate for each

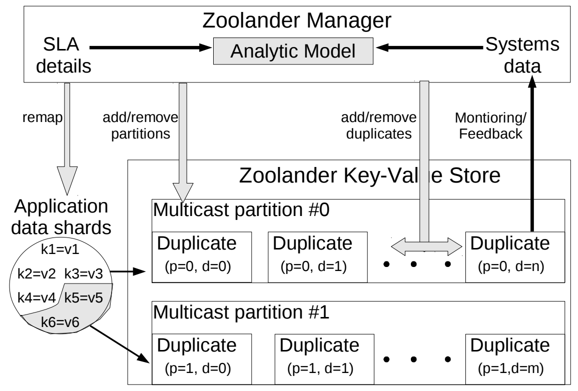


Figure 3: The Zoolander key value store. SLA details include a service level and target latency bound. Systems data samples the rate at which requests arrive for each partition, CPU usage at each node, and network delays. We use the term *replication policy* as a catch all term for shard mapping, number of partitions, and the number of duplicates in each partition.

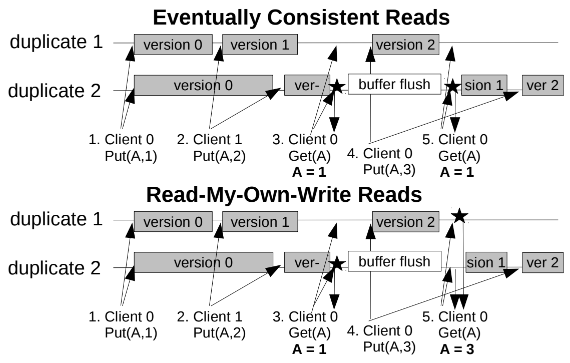


Figure 4: Version based support for read-my-own-write and eventual consistency in Zoolander. Clients funnel puts through a common multicast library to ensure write order. The star shows which duplicate satisfies a get. Gets can bypass puts.

partition and are captured by our queuing model. The third option is captured by our geometric series.

3.1 Consistency Issues

A read after write to the same key in Zoolander returns either a value that is at least as up to date as most recent write by the client (read my own write) or the value of an earlier, valid write (eventual). We can also support strong consistency funneling all accesses through a single multicast node. However, we rarely use strong consistency in any Zoolander deployments. As many prior works have noted [12, 18, 24, 39], read-my-own writes and eventual consistency normally suffice.

To support read-my-own-write consistency, each duplicate processes puts in FIFO order. Gets (reads) may be processed out of order. Clients accept reads only if the version number exceeds the version produced by their last write. For eventual consistency, Zoolander clients ignore version numbers. Figure 4 clearly depicts the supported consistency. Read my own write avoids stale data but gives up redundancy.

Propagating Writes: To ensure correct results, writes must propogate to every duplicate and every duplicate must see writes in the same order. Zoolander achieves this by using multicast. Zoolander's *client side* library keeps IP addresses for the head node of each duplicate. When client's issue a put request, the library issues D identical messages to each duplicate in a globally fixed order. In the future, we hope to replace this library with networking devices with hardware support for multicast.

Software multicast ensures that writes from a single client arrive in order, but writes from different clients can arrive out of order. We assume that multiple clients racing to update the same key is *not* the common case. As such, Zoolander provides a simple but costly solution. To share keys, clients funnel writes through a master multicast client. This approach sacrifices throughput but ensures correct results (see Figure 4).

Choosing the Right Store: By extending existing key value stores, Zoolander inherits prior work on achieving high availability and throughput. The downside is that there are many key value stores; each tailored for high throughput under a certain workload. Zoolander leaves this choice to the storage manager. In our tests, the default store is Zookeeper [18] because of its wait-free features. However, for online services that need high throughput and rich data models [7, 8, 14], we extend Cassandra [16]. We have also run tests with in-memory stores Redis and Memcached.

3.2 Implementation Issues

Overhead: Our software multicast is on the datapath of every write; It must be fast. Our multicast library avoids TCP handshakes by maintaining long-standing TCP connections between clients and duplicates. Also, Zoolander eschews costly RPC in favor of callbacks. Clients append a writeback port and IP to every access that goes through our multicast library. Duplicates respond to clients directly, bypassing multicast. We measured the maximum number of writes, read-my-own reads, and eventual reads supported per second in Zoolander with Zookeeper as the underlying store. Table 2 compares the results to the throughput of Zookeeper by itself [18]. These tests were conducted on our private cloud.

Bandwidth: Each duplicate receives the same workload and uses the same network bandwidth. At scale, duplicates could congest datacenter networks. Zoolander takes 2 steps to use less bandwidth. First, writes return only "OK" or "FAIL", not a copy of data. Second, for

Relative Throughput & Processing Overhead		
Writes	Read-my-own-write Reads	Eventual Reads
95%(48us)	94%(52us)	99%(<1us)

Table 2: Zoolander's maximum throughput at different consistency levels relative to Zookeeper's [18]. In parenthesis, average latency for multicast and callback.

reads, Zoolander re-purposes replicas set up for fault tolerance as duplicates. Such replicas are common in production [12, 39]. Figure 5(A) compares the bandwidth used by naive support for replication for predictability against Zoolander's approach. The baseline is the bandwidth consumed by a 3-node quorum system [12, 39]. Our approach lowers bandwidth usage by 2X.

Dyanamic Systems Data: Zoolander continuously collects data using sliding windows. To keep overhead low, we collect data for only a random sample of storage accesses. For each sampled access, we collect response time, service time, accessed shard number, and network latency. A window is a fixed number of samples.

We compute the mean network latency and arrival rate for each window. We use the information gain metric to determine if our CDF data has diverged. If we detect that the CDF may have diverged, we collect samples more frequently, waiting for the information gain metric to converge on new CDF data. Figure 5 demonstrates the benefits of service time windows. First, we ran our e-science workload (Gridlab-D), then we injected an additional write-only workload on the same machine, and finally we added a read-only workload also. Our sliding windows allow us to capture accurate service time distributions shortly after each injection, as shown by convergence on information gain.

Fault Tolerance: Zoolander can tolerate duplicate, partition, software multicast, and client failures. Duplicate failures are detected via TCP Keep Alive by the software multicast. Every duplicate receives every write, so between storage accesses, software multicast can simply remove any failed duplicate from the multicast list.

A partition fails when its only working duplicates fails. When this happens, Zoolander manager uses transaction logs from the last surviving duplicate to restart the partition. This takes minutes but is automated. Software multicast is a process in the client-side library. On restart, it updates its multicast list with Zoolander manager. This process takes only milliseconds. However, when software multicast is down, the entire partition is unavailable.

3.3 Model Validation & System Results

Thus far, we have developed an analytic model for replication for predictability. We have also described the system design for Zoolander, a key value store that fully supports replication for predictability at scale. Here, we show that Zoolander achieves performance expected by our model and that the model has low prediction error.

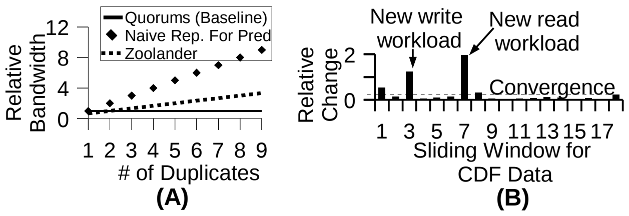
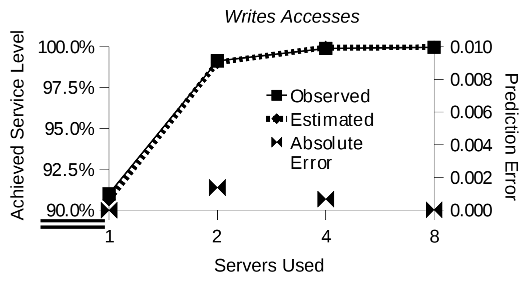


Figure 5: (A) Zoolander lowers bandwidth needs by repurposing replicas used for fault tolerance. (B) Zoolander tracks changes in the service time CDF relative to internal systems data. Relative change is measured using information gain.

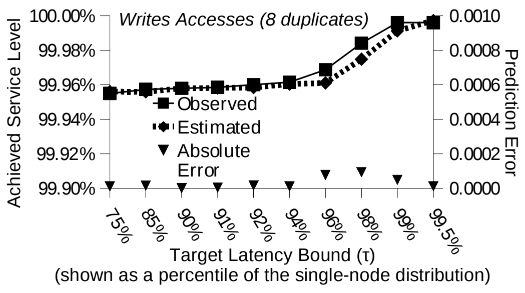
We deployed Zoolander on the private cloud described in Section 2. We used Zookeeper as the underlying key-value store. We focus on data sets that fit within memory (i.e., in-memory key-value stores backed up with local disk). We used 1 partition for these tests. We issued 1M write accesses in sequence without any concurrency. We used the 90th percentile of the collected service time distribution as the default latency bound (τ=5ms). The average response time in this setup was 3ms, so our latency bound allowed only 2ms for outliers. The SLO for Zookeeper without Zoolander was: *90% of accesses will complete within 5ms.*

We added duplicates to Zoolander one at a time, issuing the same write workload each time we scaled out. Figure 6(a) shows Zoolander's performance, i.e., achieved service level, as duplicates increase. Specifically, the achieved service level grew as duplicates were added. For example, under 8 instances, Zoolander could support the following SLO: *99.96% of write accesses will complete within 5ms.* The graph also shows that Zoolander had absolute error (i.e., actual service level minus predicted) below 0.002 in all cases. **This is a key result: Scaling out via replication for predictability strengthens SLOs without raising latency bounds.**

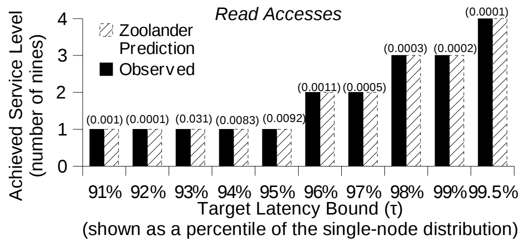
In our next test, we set the number of duplicates to 8. We used the same service time distribution from above. We then changed the latency bound (τ) to different percentiles in the single-node distribution, from the 75th to 99.5th. High percentiles led to several-nine service levels in Zoolander, forcing our model to be accurate with high precision. Low percentiles required Zoolander to accurately model more accesses. Figure 6(b) shows our model's accuracy as the latency bound increased. Absolute error was within 0.0001 for high and low percentiles. In Figure 2(a), we observed that write access times had a heavy-tail distribution that started around the 96th percentile. Figure 6(b) shows a steeper



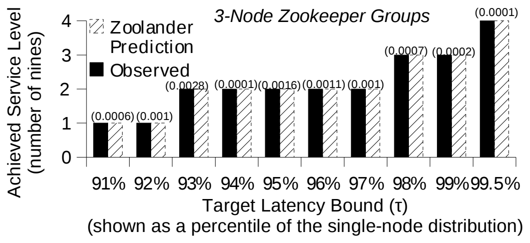
(a) Achieved service levels against Zoolander predictions as duplicates increase. Observed and estimated lines overlap.



(b) Service levels as the target latency bound changes.



(c) Service levels achieved on read-only accesses. 2 duplicates used.



(d) Service levels achieved under 3-node Zookeeper deployment. 2 duplicates used.

Figure 6: Validation Experiments

slope (strong gains) for latency bounds after the 96th percentile. For instance, setting the latency bound to the 99th percentile of single-node distribution (τ=15ms), RP Zookeeper achieved 99.991% service level using only 4 duplicates. In other words, adding duplicates scaled the service level by two orders of magnitude.

Diverse Workloads: Figures 6(c) and 6(d) shows the number of nines achieved under read accesses and under larger Zookeeper cluster size. On our cloud platform, reads completed in microseconds [18]. Sometimes our

1 node CDF	Base Geom. Series	w/ Sliding window	Full model w/ μ_{net} & μ_{rep}
60.0000	0.0835	0.0494	**0.0103**

Table 3: Percentage-point error of different versions of Zoolander's model (x100). Results for 16-node, shared L2 test.

software repeater had not finished broadcasting accesses before a duplicate finished the job. Figure 6(c) shows the results with just 2 duplicates. As we varied the latency bound, Zoolander accurately estimated service level. We focus on the number of nines because it is a common metric in practice for SLAs. Zoolander and our model agreed on the number of nines.

Figure 6(d) shows results where we set the cluster size to 3 under a write workload. Zookeeper uses an atomic broadcast to issue cluster writes. Communication within duplicate clusters increases anomalies. Despite this increase, Zoolander met our model's expectations across all tested latency bounds.

Heterogeneous Platforms: In our toughest test for Zoolander, we made a fundamental, runtime change to our cloud platform: We allowed instances to share the L2 cache. We started Zoolander with a CDF based on private L2 caches and used our continuous monitoring to discover the new CDF (window size was 10,000). We ran a total of 1M accesses. Our input latency bound (τ) was set to the 60th percentile of single-node, private-L2 service time distribution (just 3ms). A 16-instance Zoolander achieved a service level of 99.916% under this latency bound. Our full model predicted 99.927%. Table 3 shows the absolute percentage point error of different versions of the Zoolander model. The geometric series and continuous monitoring improve accuracy most.

4 Model-Driven SLO Analysis

Zoolander can scale out via replication for predictability or via partitioning. The analytic model, presented in Section 2, helps Zoolander manager choose the most efficient replication policy. The analytic model can also provide marginal analysis on the SLO achieved as key input parameters vary. Specifically, we varied the request arrival rate and used our model to predict SLO achieved. We fixed the number of nodes (4) and we fixed the systems data. We compared 3 replication policies: 1) using only replication for predictability (i.e., 1 partition with 4 duplicates), 2) using only traditional approaches (i.e., 4 partitions with 1 duplicate each), and 3) using a mixed approach (i.e., 2 partitions and 2 duplicates each). Note, our model predicts the same service levels under a k-duplicate partition with arrival rate λ as it does under N k-duplicate partitions with arrival rate $N * \lambda$, making our results relevant to larger systems.

Recall, our model is biased toward partitioning. We naively assume that each partition divides workload evenly with no internal hot spots or convoy effects. Thus, we are really comparing accurate predictions on replication for predictability to best-case predictions for partitioning. More generally, our model makes best-case predictions for any approach that reduces accesses per node by dividing work, including replication for throughput.

The results of our marginal analysis are shown in Figures 7(a–b). The y-axis in these figures is "goodput", i.e., the fraction of requests returned within SLO. The x-axis for these figures is the normalized arrival rate, i.e., the arrival rate over the maximum service rate. In queuing theory terminology, the normalized arrival rate is called system utilization. The latency bound changes across the figures. The results show arrival rates under which the studied replication heuristics excel. Specifically:

1. Zoolander's mixed approach, using both replication for predictability and partitioning, offers the best of both worlds. Replication for predictability alone increased service levels but only under low arrival rates. Partitioning alone supported high arrival rates but with low service levels. The mixed approach supported high arrival rates (>40% utilization) and achieved high SLO.

2. As the latency bound increased, replication for predictability supported higher arrival rates, and similarly, partitioning provided higher service levels.

3. Replication for predictability performs horribly under high arrival rates. Recall, all duplicates have the same queuing delay, once this delay exceeds the latency bound, replication for predictability offers no benefit. It's performance falls of a cliff.

4. Divide-the-work approaches simply can't achieve high service levels under tight latency bounds. When we set $\tau = 3.5$ms, goodput under traditional only fell below 94%. A mixed approach achieved 99% goodput.

Cost Effectiveness: SLO violations can be costly. For online e-commerce services, violations reduce sales and ad clicks. For data processing services, violations deprive business leaders of data needed to make good, profitable choices. All else being equal, reducing SLO violations means reducing costs. Replication for predictability reduces SLO violations but it uses more nodes. Nodes also cost; They use energy, their components (memory and disk) wear out, and they have management overheads. We used our model to study the cost effectiveness of using more nodes to reduce SLO violations.

We set a latency bound (τ) of 7ms and used systems data taken from our private cloud. We computed the number of SLO violations as the arrival rate changed. To provide intuition, the number of SLO violations is essentially the product of x and $(1-y)$ for (x,y) pairs in Figure 7b. The rate of violations (λ^{vio}) is shown below. F_{zk} represents our model with systems data from Zookeeper.

$$\lambda^{vio} = \lambda * (1 - F_{zk}(\tau, \lambda))$$

We used a linear model to assess cost effectiveness. Total cost was the sum of 1) SLO violations (λ^{vio}) multiplied by cost per violation (cpv) and 2) nodes used (N) multiplied by the cost per node per unit time (cpn). The model is shown below.

$$cost = N * cpn + \lambda^{vio} * cpv$$

The cost per violation and cost per node vary from service to service. We studied the relative cost between these parameters. Specifically, we set $cpn = 1$ and varied cpv, as shown in Figures 8(a-b).

We compared three replication policies. The default approach, or "do nothing", did not scale out. It used 1 1-duplicate partition ($N = 1$) and allowed SLO violations to increase with the arrival rate. The replica-

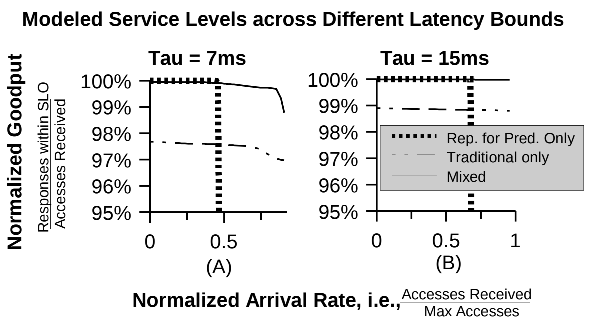


Figure 7: Trading throughput for predictability. For replication for predictability only, $\lambda = x$ and $N = 4$. For traditional, $\lambda = \frac{x}{4}$ and $N = 1$. For the mixed Zoolander approach, $\lambda = \frac{x}{2}$ and $N = 2$. Our model produced the Y-axis.

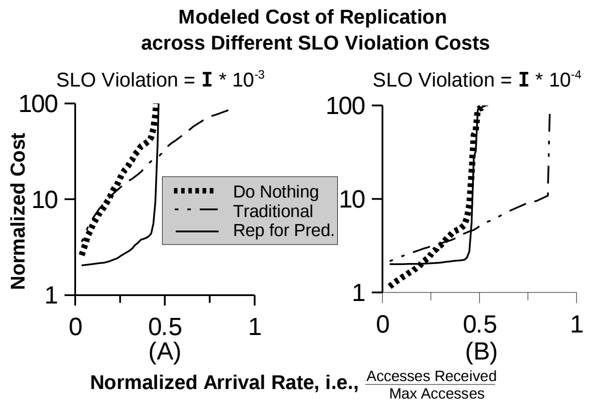


Figure 8: Cost of a 1-node system, 2 partition system, and 2 duplicate system across arrival rates. Lower numbers are better.

tion for predictability approach used 1 2-duplicate partition ($N = 2$) and reduced SLO violations under low arrival rates. The traditional approach used 2 1-duplicate partitions ($N = 2$). Note, N refers to the number of duplicates—each duplicate could comprise many nodes. We found the following insights:

1. When 100 SLO violations cost more than a node, replication for predictability is cost effective, until queuing delay exceeds the latency bound and service levels fall of the cliff.

2. If SLO violations are cheap, e.g., a node costs more than 10,000 violations, replication for predictability is never cost effective, even under low arrival rates.

3. If arrival rates change, the most cost effective approach will also change. When SLO violations are neither cheap nor expensive, all three approaches can be cost effective under certain rates.

The exact cost of an SLO violation depends on the service. Online services have found ways to compute *cpv* for their workloads. It is harder to compute *cpv* in emerging services, e.g., Twitter trend analysis or smart-grid power management. In these services, violations map only indirectly to revenue. However, if such violations lead to stale results that lead to poor decisions, the real cost of such violations can be very high.

5 Zoolander in Action

For this section, we studied Zoolander under intense arrival rates, e.g., workloads produced by online services. These tests used up to 144 Amazon EC2 units. EC2 is widely used by e-commerce sites and web portals. It's prices are well known. Our goal was to compare Zoolander scaling strategies and to highlight real world settings where replication for predictability is cost effective.

Many online services see diurnal patterns in the arrival rates of user requests [4, 34]. Request arrival rates can fall by 50% between 12am–4am compared to daily peaks between 9am–7pm. As a result, fewer nodes are needed in the night than in the day time. Nonetheless, services must buy enough nodes to provide low response times under peak arrival rates. Some services save energy by using only a fraction of their nodes during the night, turning off unused nodes. However, in datacenters, energy costs are low at night (because demand for electricity is low). Nighttime energy prices below \$0.03 are common. The typical service would save only \$0.12 per night by turning off a 1KW server during this period. Zoolander can exploit underused nodes in a different way; Turning them into duplicates to reduce SLO violations.

We compared the opportunity costs of reducing SLO violations against turning off machines. Figure 9 shows the competing replication policies. During the daytime, each node is needed for high throughput and operates under its max arrival rate. However, at nighttime, the arrival rate drops by 50%, allowing us to place 2 shards on 1 node or to use replication for predictability. To save energy at night, our replication policy consolidates shards, using as few nodes as possible without exceeding the peak per-node arrival rate. Our workload accesses all shards evenly, i.e., no hot spots.

Replication for predictability can be applied naively on top the energy saving approach by using idle nodes as duplicates. SCADS manager adopted this approach [39]. However, our findings in Section 4 suggest that arrival rates on nodes that use replication for predictability should be low. We decided to use replication for predictability more sparingly, keeping arrival rates low for the duplicates. For every 6 nodes, we placed 4 shards on 2 nodes (like in the energy saving approach). The remaining 4 nodes hosted 2 shards via 2 2-duplicate partitions. Our approach had 9.7% fewer violations compared to the naive approach described above.

To make the test realistic, we setup Zoolander on EC2 and tried to mimic the scale of TripAdvisor's workload. Public data [14] shows that TripAdvisor receives 200M user requests per day. On average, each user request accesses the back-end Memcached store 7 times, translating to 1.4B storage accesses per day. Learning from recent studies, we assumed the arrival rate would drop by 50% [4]. Our goal was to support 29M accesses per hour.

We used 48 clients that issued a mix of 15% Gets and 85% Puts across 96 shards. Gets/Puts were issued in batches of 20, reflecting correlated storage accesses within user requests. Each batch arrived independently, leading to exponentially distributed inter-arrival times. Note, our clients followed a realistic, open-loop workload model. Duplicates in these tests were 1-node Cassandra [16]. In a 4 hour test, our clients issued over 160M key-value lookups (40M per hour).

During our tests, Zoolander achieved high throughput and fault tolerance. While these metrics do not reflect Zoolander's contribution, they are not weak spots either! To support 40M lookups per hour, Zoolander used 48 EC2 compute units with Cassandra as the underlying key-value store. Peak throughput was 431 lookups per second per EC2 unit, about 20% higher than the average achieved by Netflix operators [7]. We encountered 546 whole partition failures across the 144 nodes where either Cassandra or the software multicast crashed. During those failures, 2,929 lookups failed. Multicast, duplicates or callbacks caused 1,200 of those failed lookups.

SLO violations also occur when Zoolander migrated data to its nighttime setup. Migrations periods are shown in Figure 10(a). The figure is based on a trace from [34]. Moving to from the daytime setup to Zoolander's night-

node id	daytime setup	night time energy saver	night time Zoolander
0	accesses/hr = 1.6M hosted shard(s) = A	accesses/hr = 1.6M hosted shard(s) = A,B	accesses/hr = 1.6M hosted shard(s) = A,B
1	accesses/hr = 1.6M hosted shard(s) = B	accesses/hr = 1.6M hosted shard(s) = C,D	accesses/hr = 1.6M hosted shard(s) = C,D
2	accesses/hr = 1.6M hosted shard(s) = C	accesses/hr = 1.6M hosted shard(s) = E,F	accesses/hr = 0.8M hosted shard(s) = E
3	accesses/hr = 1.6M hosted shard(s) = D	NOT USED	accesses/hr = 0.8M hosted shard(s) = E
4	accesses/hr = 1.6M hosted shard(s) = E	NOT USED	accesses/hr = 0.8M hosted shard(s) = F
5	accesses/hr = 1.6M hosted shard(s) = F	NOT USED	accesses/hr = 0.8M hosted shard(s) = F

Figure 9: Replication strategies during the nighttime workload for an e-commerce service.

time setup needed 4 shard migrations(see Figure 9). Moving back to daytime setup needed 2 more migrations. Zoolander used existing techniques for shard migration. We measured migration-induced violations under load and added these to Zoolander's costs.

We used the cost model in Section 4 to compare the nighttime replication policies in Figure 9. We studied two different cost per node settings. In the *private cloud* setting, the cost of a node is a function of its energy usage only. We assumed a cost of \$0.03KWh and that each node (an EC2 unit) used 100W, thus $cpn = \$0.003$. In the public cloud setting, the cost of a node includes everything provided by EC2 (i.e., high availability, EBS, etc). As of this writing, cpn of a small EC2 unit was \$0.085 (20X more than energy only costs). The energy saving approach used 3 nodes, whereas the Zoolander approach used 6. We set the SLO latency bound (τ) for a batch of lookups to 150ms. Out of 160M requests the Zoolander approach incurred only 57K SLO violations compared to 85K in the energy saving approach—a reduction of 32%.

Figure 10(b) plots relative cost as a function of cost per 1000 violations (cpv). Lower numbers are better for Zoolander. The x-axis is log scale base 10. As SLO costs increase, the Zoolander approach becomes more cost effective. Without considering migration costs, the relative cost converges to 68% quickly in the private cloud setting where cpn is very low. Migration costs increase relative cost by 16%, but Zoolander remains highly cost effective in private clouds. Figure 10(b) shows that Zoolander spends \$0.79 to every dollar spent by energy saver approach, saving 21%. The public cloud setting requires higher cpv to be cost effective.

In their fiscal statement for the 4^{th} quarter of 2011, TripAdvisor earned \$122M from click- and display-based advertising. We divided this number by 200M daily user requests to get revenue per 1,000 page views of \$6.81. Using prior research, we estimated that each SLO violation (a 100ms delay) would lead to a 1% loss in prof-

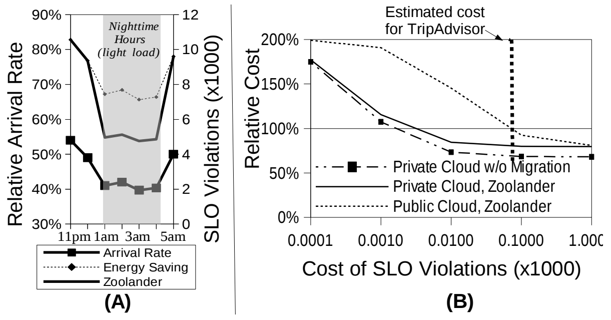


Figure 10: (A) Zoolander reduced violations at night. From 12am-1am and 4am-5am, Zoolander migrated data. We measured SLO violations incurred during migration. (B) Zoolander's approach was cost effective for private and public clouds. Relative cost is ($\frac{\text{zoolander}}{\text{energy saver}}$)

its [30], meaning $cpv = \$0.068$. Under this setting, the Zoolander approach was cost effective for private settings and broke even with the energy savings approach under public cloud settings. When we consider migration costs for the energy savings approach, Zoolander is cost effective even for public clouds.

Model-Driven Management The nighttime policy for the EC2 tests was a heuristic based on insights from Section 4. Heuristics derived from principled models underlie many real world systems. Alternatively, Zoolander's model can be queried directly to find good policies.

We used systems data from Zookeeper and set τ to 3.5ms, a very low latency bound. We studied the hourly arrival rates (λ) shown in Table 4. For each rate, our model computed the expected SLO under 8 policies: 8 partitions(p) each with 1 duplicate(d), 4p with 2d, 2p with 4d, 6p with 1d, 3p with 2d, 2p with 3d, 4p with 1d, and 2p with 2d. Table 4 shows the policy that met SLO using the fewest nodes. The 5 policies shown all differ, including policies with more than 2 duplicates.

Target SLO:	*98% of accesses complete in 3.5ms*				
Accesses/Hour:	2K	850K	1M	1.5M	1.9M
Best Policy:	4p/1d	2p/2d	2p/3d	3p/2d	4p/2d

Table 4: Best replication policy by arrival rate

6 Related Work

Zoolander improves response times for key value stores by masking outlier access times. Contributions include: 1) a model of replication for predictability that is blended with queuing theory, 2) full, read-and-write support for replication for predictability, and 3) experimental results that show the model's accuracy and cost effective application of replication for predictability. Related work falls into the categories outlined below.

Replication for predictability and cloning: Google's BigTable re-issues storage accesses whenever an initial access times out (e.g., over 10ms) [9, 10]. Outliers will rarely incur more than 2 timeouts. This approach applies replication for predictability only on known outliers, reducing its overhead compared to Zoolander. Writes present a challenge for BigTable's approach. If writes that are not outliers are sent to only 1 node, duplicates diverge. If instead, they are sent to all nodes re-issued accesses would not mask delays because they would depend on slow nodes. Zoolander avoids these problems by sending all writes to all replicas.

SCADS revived replication for predictability, noting its benefits for social computing workloads [3]. SCADS sent every read to 2 duplicates [39] and supported read-only or inconsistent workloads. Replication for predictability strengthened service levels by 3–18%. Zoolander extends SCADS by scaling replication for predictability, modelling it, and supporting consistent writes. Section 5 showed that, as arrival rates increase, our model can find replication policies that outperform the fixed 2-duplicate approach.

Data-intensive processing uses cloning to mask outlier tasks. Early Map-Reduce systems cloned tasks when processors idled at the end of a job [11]. Mantri et al. [2] used cloning throughout the life of a job to guard against failures. In both cases, the number of duplicates were limited. Also, map tasks issue only read accesses. Recent work used cloning to mask delays caused by outlier map tasks [1], providing a topology-aware adaptive approach to save network bandwidth. Like Zoolander, this work focused on cost effective cloning. Zoolander's model advances this work, allowing managers to understand the effect of budget policies in advance. Another recent work [21] sped up data-intensive computing via replication for predictability. This work defines budgets in terms of reserve capacity and uses recent models on map-reduce performance [41].

Adaptive partitioning and load balancing: Heavy tail access frequencies also degrade SLOs. Hot Spots are shards that are accessed much more often than typical (median) shards. Queuing delays caused by hot spots can cause SLO violations. Further, hot spots may shift between shards over time. SCADS [39] threw hardware at the problem by migrating the hottest keys within a shard via partitioning and replication. Other works have extended this approach to handled differentiated levels of service [33] and also for disk based systems [27]. Consistent hashing provides probabilistic guarantees on avoiding hot spots [36, 42]. [19] extends consistent hashing by wisely placing data for low cost migration in the event that a hot spot arises. Locality aware placement can also reduce the impact of hot spots [23].

Both replication for predictability and power-of-two load balancing [28] involve sending redundant messages to nodes. However, in load balancing, the nodes do not share a consistent view of data. Just-idle-queue load balancing includes a related sub problem where an idle node must update exactly 1 of many queues [25]. Here, taking the smallest queue is like taking the fastest response in replication for predictability and reduces heavy tails.

Removing performance anomalies: Background jobs are not the only root cause of heavy tails, performance bugs that manifest under rare runtime conditions also degrade response times. Removing performance bugs requires tedious and persistent effort. Recent research has tried to automate the process. Shen et al. use "reference executions" to find low level metrics affected by bug manifestations, e.g., system call frequency or pthread events [32]. These metrics uncovered bugs in the Linux kernel. Attariyan et al. used dynamic instrumentation to find bugs whose manifestation depended on configuration files [5]. Recent works have found bugs at the application level [22, 40]. Debugging performance bugs and masking their effects, as Zoolander does, are both valuable approaches to make systems more predictable, but neither is sufficient. Some root causes, like cache misses [4], should be debugged. Whereas, other root causes manifest sporadically but, if they were fixed, could unmask bigger problems [35].

The operating system and its scheduler are a major reason for heavy tails. Two recent studies reworked memcached, removing the operating system from the datapath via RDMA [20, 37]. While many companies can not run applications like memcached outside of kernel protection, these studies suggest that the OS should be redesigned to reduce access-time tails.

7 Conclusion

This paper presented Zoolander, middleware that fully supports replication for predictability on existing key value stores. Replication for predictability redundantly sends each storage access to multiple nodes. By doing so, it sacrifices throughput to make response times more predictable. Our analytic model explained the conditions where replication for predictability outperforms traditional, divide-the-work approaches. It also provided accurate predictions that could be queried to find good replication policies. We tested Zoolander with Zookeeper and Cassandra. Its overhead was low. Our largest test (spanning 144 EC2 compute units) showed that Zoolander achieved high throughput and strengthened SLOs. By wisely mixing scale-out approaches, Zoolander reduced operating costs by 21%.

Acknowledgements: Thanks to Arif Merchant, John Douceur, Jian Yin, Kai Shen, and Daiyi Yang for early feedback. Nan Deng and Jaimie Kelley provided valuable feedback on Zoolander's interface. This work was funded by NSF EAGER 1230776.

References

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, 2013.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.

[3] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna. Scads: Scale-independent storage for social computing applications. 2009.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, 2012.

[5] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX OSDI*, 2012.

[6] P. Bailis. Doing redundant work to speed up distributed queries. `http://www.bailis.org/blog/`.

[7] A. Cockcroft and D. Sheahan. Benchmarking cassandra scalability on aws - over a million writes per second. `http://techblog.netflix.com`, Nov. 2011.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[9] J. Dean. Achieving rapid response times in large online services, 2012.

[10] J. Dean and L. Barroso. The tail at scale. 2013.

[11] J. Dean and S. Gemawat. Mapreduce: simplified data processing on large clusters, Dec. 2004.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. In *ACM SOSP*, 2007.

[13] J. Dike. User-mode linux.

[14] A. Gelfond. Tripadvisor architecture - 40m visitors, 200m dynamic page views, 30tb data. `http://highscalability.com`, June 2011.

[15] J. Gray. *Transaction Processing: Concepts and Techniques*. 1993.

[16] E. Hewitt. Cassandra: The definitive guide, 2011.

[17] S. Hsiao, L. Massa, V. Luu, and A. Gelfond. An epic tripadvisor update: Why not run on the cloud? the grand experiment. `http://highscalability.com/blog/2012/10/2/`.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[19] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *IEEE ICAC*, 2013.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *ACM SOCC*, 2012.

[21] J. Kelley and C. Stewart. Balanced and predictable networked storage. In *International Workshop on Data Center Performance*, 2013.

[22] M. Kim, R. Sumbaly, and S. Shah. Root cause detection in a service-oriented architecture.

[23] M. Kozuch, M. Ryan, R. Gass, S. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. Lpez, M. Stroucken, and G. R. Ganger. Tashi: Location-aware cluster management. In *First Workshop on Automated Control for Datacenters and Clouds (ACDC'09)*, 2009.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP*, Cascais, Portugal, Oct. 2011.

[25] Y. Lu, Q. Xie, G. Kilot, A. Geller, J. Larus, and A. Greenburg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. In *PERFORMANCE*, 2011.

[26] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007.

[27] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: quality-of-service in large disk arrays. In *IEEE ICAC*, 2011.

[28] M. mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[29] J. C. Mogul. Tcp offload is a dumb idea whose time has come. In *HotOS*, 2003.

[30] rigor.com. Why performance matters to your bottom line. `http://rigor.com/2012/09/roi-of-web-performance-infographic`.

[31] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent power. In *ACM ASPLOS*, Mar. 2011.

[32] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, 2009.

[33] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage, 2012.

[34] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys Conf.*, Mar. 2007.

[35] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE MASCOTS*, 2010.

[36] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan.

[37] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[38] TripAdvisor Inc. Tripadvisor reports fourth quarter and full year 2011 financial results, Feb. 2012.

[39] B. Trushkowsky, P. Bodk, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *USENIX FAST*, 2011.

[40] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *ACM SIGMETRICS*, 2012.

[41] Z. Zhang, L. Cherkasova, A. Verma, and B. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *IEEE ICAC*, Sept. 2012.

[42] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash by using less cache. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2012.

Preemptive ReduceTask Scheduling for Fair and Fast Job Completion

Yandong Wang[*] *Jian Tan*[†] *Weikuan Yu*[*] *Li Zhang*[†] *Xiaoqiao Meng*[†]

Auburn University[*] *IBM T.J Watson Research*[†]
{wangyd,wkyu}@auburn.edu {tanji,zhangli,xmeng}@us.ibm.com

Abstract

Hadoop MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks among data-intensive applications. However, under this scheme, Hadoop schedulers do not work effectively for both phases. We reveal that there exists a serious fairness issue among jobs of different sizes, leading to prolonged execution for small jobs, which are starving for reduce slots held by large jobs. To solve this fairness issue and ensure fast completion for all jobs, we propose the *Preemptive ReduceTask* mechanism and the *Fair Completion scheduler*. Preemptive ReduceTask is a mechanism that corrects the monopolizing behavior of long reduce tasks from large jobs. The Fair Completion Scheduler dynamically balances the execution of different jobs for fair and fast completion. Experimental results with a diverse collection of benchmarks and tests demonstrate that these techniques together speed up the average job execution by as much as 39.7%, and improve fairness by up to 66.7%.

1 Introduction

MapReduce [10] is a simple yet powerful programming model that is increasingly deployed at many data centers for the analysis of large volumes of unstructured data. Hadoop [1] is an open-source implementation of MapReduce. It divides a MapReduce job into two types of tasks, map tasks (MapTasks) and reduce tasks (ReduceTasks), and assigns tasks to multiple workers called TaskTrackers for parallel data processing.

To support many users and jobs (large batch jobs and small interactive queries), Hadoop MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks for data-intensive applications. The Hadoop Fair Scheduler (HFS) [4] and Hadoop Capacity Scheduler (HCS) [3] have focused on fairness among MapTasks. These schedulers strive to maximize the use of system capacity and ensure fairness among different jobs. However, they do not work effectively for both phases. What complicates the matter is the distinct execution behaviors of MapTasks and ReduceTasks. Unlike MapTasks which are launched one group after the other to process data splits, ReduceTasks have a different execution pattern. Once a ReduceTask is launched, it occupies the reduce slot until completion or failure.

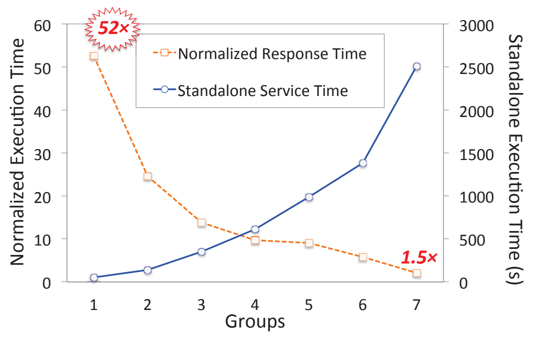


Figure 1: Unfair Execution among Different Size Jobs

We have examined the performance of Hadoop schedulers using a synthetic workload of jobs submitted to a shared MapReduce cluster. Jobs are divided into 7 groups based on their increasing data sizes; jobs in the same group are identical. They arrive according to a Poisson random process. Figure 1 shows the comparison of the *normalized execution time*, which is defined as the ratio between a job's actual execution time and its standalone execution time (the time when a job is running in the system alone). As shown in the figure, the standalone execution time of jobs in each group increases in proportion to their input data size. However, the completion of these jobs varies dramatically with HFS. Jobs in the smaller groups have much worse normalized execution times, indicating that they must wait very long (as much as 52× longer than the stand-alone execution time). Such scheduling behavior contradicts users' intuitive expectation that smaller jobs should be completed faster and turned around more quickly.

To address this fairness issue and ensure fast completion for jobs of various sizes, we design a combination of two techniques: the *Preemptive ReduceTask* mechanism and the *Fair Completion Scheduler*. Preemptive ReduceTask is a solution to correct the monopolizing behavior of long ReduceTasks. By enabling a lightweight working-conserving option to preempt ReduceTasks, Preemptive ReduceTask offers a mechanism to dynamically change the allocation of reduce slots. On top of this preemptive mechanism, the Fair Completion Scheduler is designed to allocate and balance the reduce slots among jobs of different sizes. In summary, we make the following contributions on the scheduling of jobs in data centers for fair and fast job completion.

- We examine the unfairness issue of MapReduce jobs execution in detail and identify the key shortcomings of existing schedulers in balancing the allocation of reduce slots among jobs.
- We introduce the Preemptive ReduceTask mechanism for lightweight, work-conserving preemption, on top of which we design the Fair Completion Scheduler that improves both the fairness and execution of MapReduce jobs.
- We have conducted a systematic evaluation of Fair Completion Scheduler. Our results demonstrate that it can reduce the average execution time of workloads by up to 39.7% and improves the fairness by as much as 66.7%, when compared to HFS.

2 Background and Motivation

In this section, we first provide a brief overview of Hadoop job scheduling, then discuss the issues within existing schedulers.

2.1 Job Scheduling in Hadoop

In Hadoop, the JobTracker assigns available map and reduce slots separately to jobs in the queue, one slot per task. Figure 2 shows an example of scheduling three jobs (represented by shaped blocks in three colors) on a system with three reduce slots and five map slots. The scheduling policy is based on the Hadoop Fair Scheduler. A job when running alone can satisfy its needs with all reduce slots, but it has to share the slots when other jobs arrive. Once granted a slot, a ReduceTask has to fetch data produced by all MapTasks before it completes. In the figure, Job 1 first arrives by itself. It grabs 3 map slots and 2 reduce slots for itself and completes execution. Job 2 then takes the rest of map and reduce slots. When Job 2 needs more map or reduce slots, it has to share, because Job 3 has arrived.

Each map output file has a partition for every ReduceTask, the current Hadoop scheduler greedily launches as many ReduceTasks as permitted for each job to maximize the chance of overlapping the shuffling of available intermediate data with the execution of future MapTasks. Hadoop also allows a configuration parameter

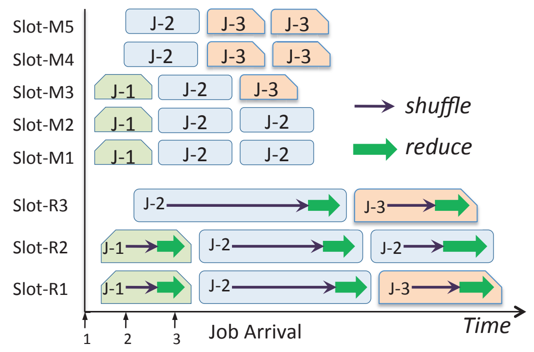


Figure 2: An Example of Scheduling Slots among Jobs

(*slowstart.completed.maps*) to delay the launch of ReduceTasks so that small jobs after large jobs can have chances to share the reduce slots.

2.2 Profiling of Unfair Slot Allocation

To closely examine the fairness issue between different jobs, we conduct an experiment on a cluster of 20 nodes. 40 map slots are created on 10 nodes, and 20 reduce slots on the other 10 nodes. 8 jobs are sequentially submitted into the cluster every 60 seconds. Job 3 is a large job that requires 20 ReduceTasks. Figure 3 shows the usage of map and reduce slots by 8 jobs. Map slots are shared among jobs over time as jobs arrive and leave, but reduce slots are all occupied by Job 3. As a result, Jobs 4-8 cannot get a share until Job 3 completes, even if they have successfully finished all their MapTasks. On average, Jobs 4-8 are significantly delayed compared to their stand-alone execution times. This reveals that Hadoop Fair Scheduler is not able to achieve fair normalized execution times for all jobs. A similar behavior was also reported by an IBM study [15]. Note that there exists a dramatic variance among the normalized execution time for different jobs in the same pool and in different tests (c.f. Figure 1 and Figure 3). More importantly, when the generation rate of intermediate data is low, even if long running ReduceTasks are occupying the slots, they do not efficiently utilize the resources, and ReduceTasks periodically enter into the idle state, causing severe resource underutilization. In this experiment, on average, during 87.6% of Job 3's ReduceTasks execution time, CPUs and disks are idle and waiting for the intermediate data, and network is highly underutilized.

2.3 Proposed Solutions

The monopolizing behavior of ReduceTasks has been documented earlier as a reason to cause small jobs starve for reduce slots [17, 15, 18]. Hadoop provides a slowstart configuration option that can delay the launch of ReduceTasks and mitigate this situation, but at the cost of slowing down the shuffle phase, thus it can significantly prolong the execution times of small jobs. Zaharia *et al.* [18]

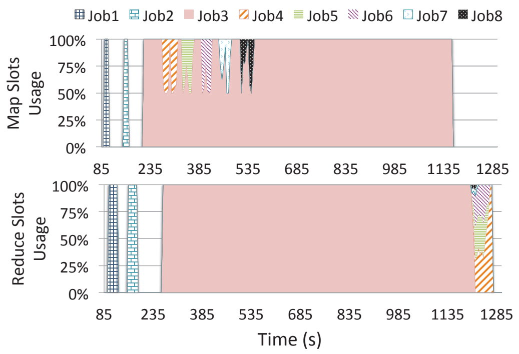


Figure 3: Run-Time Allocation Profile of Map and Reduce Slots.

proposed a copy-compute splitting mechanism, but it does not fully resolve this issue. Tan *et al.* [15] proposed a coupling scheduler to launch reducers gradually by coupling the progresses of map and reduce tasks in the same job. With this scheduler, a large job can spare reduce slots for other jobs when its map phase has not progressed much. But when a large job finishes its map phase, it still takes all available reduce slots and causes the starvation of small jobs. Like the slowstart option, the coupling scheduler delays and mitigates the monopolization of reduce slots by large jobs. But it does not solve the monopolization, instead let it progressively happen.

In this study, we examine the job fairness and efficiency issues in data centers and investigate the feasibility of lightweight task preemption and automated preemptive scheduling policy for fair and fast job completion under MapReduce context. Two techniques are designed accordingly to tackle these issues: the Preemptive ReduceTask mechanism and the Fair Completion Scheduler. Preemptive ReduceTask allows ReduceTasks to be preempted in a work-conserving manner (without losing previous I/O or computation work, or causing high overhead) during shuffle or reduce phases. The Fair Completion Scheduler builds on top of preemptive ReduceTask to automatically monitor job progresses and dynamically balance the usage of reduce slots, thereby speeding up the execution of small jobs and ensuring fairness among a large number of jobs on a shared Hadoop cluster.

3 Preemptive ReduceTask

A preemptive mechanism needs to be efficient and lightweight so that it can react fast enough to dynamic system workloads. But a ReduceTask often consumes the bulk of processing time due to its main responsibilities of fetching and merging intermediate data from all MapTasks and performing user-defined reduce computation on the merged data. In this section, we introduce our Preemptive ReduceTask mechanism that can preempt a ReduceTask at any time during its execution, with low overhead and negligible delay to the job progress.

3.1 Work-Conserving Self Preemption

Preemption is usually an OS utility to threads and processes running on a system. Operating systems such as Linux are equipped with a sophisticated thread/process table along with virtual memory to record the progresses of threads/processes and support lightweight preemption. However, there is no such utility in Hadoop to keep the ReduceTask around as a process after its preemption. Although Hadoop currently provides a *killing* based preemption mechanism, our results show that killing is a poor preemption option that can significantly delay the progress of entire job. A naive checkpoint/start mechanism is also not suitable because it dumps all memory of a ReduceTask (it can be several GB) to persistent storage and incurs very high costs. Instead we introduce a work-conserving self preemption mechanism. When requested, a ReduceTask will conserve its work and then preempt itself, i.e., exit and release reduce slot. Note that our preemptive ReduceTask keeps current APIs of Hadoop and HDFS [14] intact, all existing Hadoop applications can still function without any modification.

During the shuffle phase, a ReduceTask fetches all the segments that belong to it from all intermediate map outputs. According to the sizes of the segments, ReduceTask stores them either to local disks or in memory. Meanwhile, multiple merging threads merge fetched segments into larger segments and store them to the persistent storage. During the reduce phase, a ReduceTask organizes all the segments in a *Minimum Priority Queue* (MPQ, which has a heap structure), in which the segment that has the minimum first <key,value> pair is positioned at the head of MPQ. As the reduce phase progresses, <key,value> pairs are continually popped out from the MPQ and supplied to the reduce function.

3.1.1 Preemption during Shuffle Phase

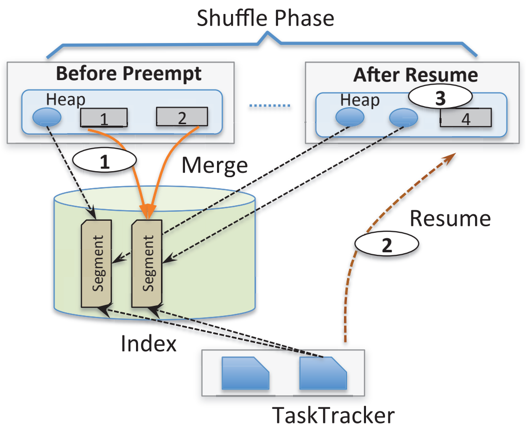


Figure 4: Preemption during Shuffle Phase

Figure 4 shows our design of work-conserving preemption when a ReduceTask is in the shuffle phase. Before preemption, a ReduceTask has a mixture of one on-disk segment and two in-memory segments, organized in a heap. Preserving the state of shuffle phase is to keep track of the shuffling status of all segments. Upon receiving a preemption request, this ReduceTask merges the in-memory segments and flushes the results to the disks (Step 1) while leaves on-disk segments untouched. The parent TaskTracker maintains an index record on the locations of fetched segments, one per preempted ReduceTask. Then the ReduceTask preempts itself and releases the slot. When the ReduceTask is later resumed (Step 2), it retrieves the index record from the parent TaskTracker, then restores the heap structure before the preemption. After that, this ReduceTask continues to fetch the rest segments from remaining map outputs (Step 3).

3.1.2 Preemption during Reduce Phase

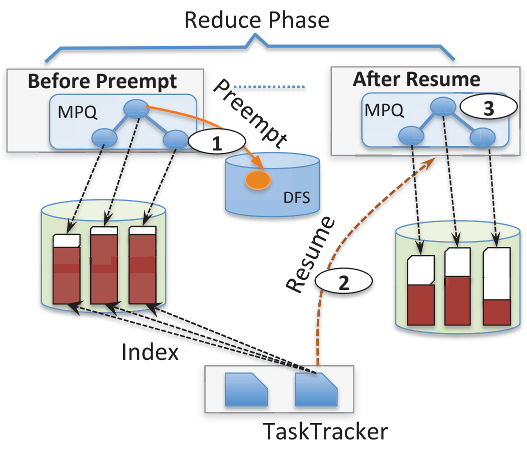


Figure 5: Preemption during Shuffle Phase

To conserve the work before preemption in the reduce phase, a ReduceTask needs to store the current results to HDFS besides recording the positions of input segments in the MPQ. In other words, ReduceTask needs to preserve the state of reduce computation at the end of each intermediate <key,val> pair, and remember the index of the last intermediate <key,val> pair at the time of preemption. Figure 5 shows our strategy for work-conserving preemption during the reduce phase. A ReduceTask is drawing <key,val> pairs from the MPQ that consists of three segments. When it receives a preemption request, it stops the reduce computation at the boundaries of <key,val> pairs (Step 1). Available results for previous <key,val> pairs are stored to HDFS. The parent TaskTracker again helps in this process by storing an index record for a preempted ReduceTask, and later provides it for preempted ReduceTask to resume its execution (Step 2). After resumption, the ReduceTask restores the MPQ again and proceeds further from the next <key,val> pair without any loss or repetition of reduce computation and intermediate data reshuffling (Step 3). During this process, to allow multiple preempted/resumed ReduceTasks to write to HDFS, we let the TaskTracker maintain the output streams to HDFS, therefore they can be shared by many ReduceTasks. Only the last ReduceTask closes the stream. In addition, Task migration is also possible for a preempted ReduceTask but it requires data to be re-fetched over the network.

4 Fair Completion Scheduler

Algorithm 1 FCS: Selecting ReduceTask to Preempt

```
1:  $L_{running}$: {a list of running jobs of decreasing remaining work.}
2:  $J_i$: {a job requesting new reduce slots.}
3:  Demand($J_i$) ← {$J_i$'s demand for reduce slots.}
4:  **if** $Available\_reduce\_slots <$ Demand($J_i$) **then**
5:      $m \leftarrow$ Demand($J_i$) - $Available\_reduce\_slots$
6:      **for all** $j \in L_{running} \wedge$ IsPreemptable($J_j$) $\wedge (m > 0)$ **do**
7:          **if** $(J_j.T_{rs} > J_i.T_{rs}) \vee ((J_j.T_{rs} == J_i.T_{rs}) \wedge (J_j.R_{left} > J_i.R_{left})$ **then**
8:              $RL_n \leftarrow$ {$J_j$'s list of running ReduceTasks}
9:              **for all** $r \in RL_n \wedge (m > 0)$ **do**
10:                 preempt $r$
11:                 $m \leftarrow m - 1$
12:             **end for**
13:         **end if**
14:     **end for**
15: **end if**
```

To efficiently balance the reduce slots among a large number of jobs of different sizes, we introduce a novel preemptive ReduceTask scheduling policy based on the remaining ReduceTasks workload of all the jobs. MapTasks are scheduled under independent scheduling policies, such as max-min fair sharing, or FLEX [17]. Because of its benefits in achieving fairness for jobs of different sizes (c.f. Section 5), we refer to it as Fair Completion Scheduler (FCS).

As a preemptive scheduler, FCS must be equipped with two algorithms: one to automatically select a ReduceTask to preempt and the other to select a ReduceTask to launch. We first describe the selection policy for preemption. To select a suitable ReduceTask and achieve fair execution, we need to evaluate the run-time progress of jobs. However, the relative progress and the remaining processing time of ReduceTasks are not available before they start. We choose the following approximations to estimate the progress.

Remaining shuffle time: This is estimated as T_{rs} through the function: $T_{rs} = (\frac{M_{left}}{M_{rate}}) \times T_{mavg}$, where M_{left} stands for the number of remaining MapTasks, M_{rate} is the average rate in completing MapTasks, and T_{mavg} is the average execution time of MapTasks that have completed or in progress. As a job makes progress in its execution, we dynamically update M_{rate} accordingly.

Remaining reduce data: This is estimated as R_{left} through the function: $R_{left} = R_{total} - R_{done}$, where R_{total} stands for the total intermediate data to reduce, and R_{done} the data that has been reduced. The latter is available during the progress of reduce phase, and the former is available when the reduce phase starts.

Execution Slackness: This is estimated as E_{slack} through the function: $E_{slack} = \frac{T_{total}}{T_{est}}$, where T_{total} is a ReduceTask's total execution time since its beginning and T_{est} is its estimated execution time based on its progress without preemption. We calculate it as $T_{est} = \frac{T_{svc}}{C_{pctg}}$, where T_{svc} is the actual execution time excluding preemption and C_{pctg} is the percentage of completed work.

FCS is designed with policies to balance reduce slots between small jobs and large jobs. It compares a job j that has the largest amount of remaining work to a job i requesting reduce slots, as shown in Line 7 of Algorithm 1. Job j's ReduceTasks are preempted if it has more work than Job i (Line 10). Essentially, this allows small jobs to preempt large jobs, solving the monopolizing behavior of long-running jobs and reducing the delay of small jobs. On the other hand, we monitor the *execution slackness* of a ReduceTask since its beginning. If its execution slackness has reached a configurable upperbound (5 by default), a ReduceTask will not be preemptable, i.e. *IsPreemptable* returns false. This enables large jobs with an option to escape preemption–keeping their reduce slots–and avoid starvation. Note that the execution slackness is a calculated number at run-time, which offers a better choice than a static parameter, for example, the number of times a ReduceTask can be preempted. Its sole purpose is to guarantee that a long job would not get seriously delayed because of frequent preemptions by other jobs. Besides taking into account of execution slackness, we avoid preempting a newly launched ReduceTask or a ReduceTask whose progress has gone over 70% to avoid overhead.

Then we describe briefly the policy for selecting a ReduceTask to launch, which is shown as Algorithm 2. In making this selection, FCS favors the jobs with the least amount of remaining work as shown in Line 2 of Algorithm 2. Jobs are firstly sorted according to their T_{rs} values, when two T_{rs} values are equal, they are sorted according to R_{left}. In addition, it takes the data locality into account, trying to launch a preempted ReduceTask on the same node that it has executed before (Line 4). A preempted ReduceTask that cannot achieve data locality will be delayed (Line 16). However, if a preempted ReduceTask has been delayed for too long because it is not able to resume on its previous node (Line 11), then FCS migrates it to another node that has available reduce

Algorithm 2 FCS: Selecting ReduceTask to Launch

```
1:  {Receiving a heartbeat from node $n$ with an empty slot.}
2:  $L_{rem}$: {a sorted list of jobs of increasing remaining work.}
3:  for all $j \in L_{rem}$ do
4:    if (Task $r$ is j's reduce task either preempted from n or never launched) then
5:      $r.migration = 0$
6:      launch $r$ on $n$
7:      return
8:    end if
9:    $T_{prt} \leftarrow$ {j's preempted ReduceTasks (oldest first)}
10:   for all $r \in T_{prt}$ do
11:     if $r.migration >$D then
12:       migrate $r$ to $n$
13:       $r.migration = 0$
14:       return
15:     end if
16:     $r.migration$ += 1
17:   end for
18: end for
```

slots (Line 12). In this algorithm, D is an approximation of $-M \times ln(\frac{1-L}{1+(1-L)})$, a similar parameter employed in the delay scheduling [19], where M is the number of nodes in the cluster and L is the expected data locality. For example, on a cluster of 20 nodes, with the expected data locality $L = 0.95$, then $D \approx 61$. With this algorithm, we fit the same delay scheduling policy (and its parameter D) nicely into FCS, and delay the launching of a ReduceTask for a future possibility to resume it on the node it was preempted, i.e., better locality. This parameter allows us to consider the tradeoff between the need of resuming ReduceTask for data locality and the need of migrating ReduceTasks for free slot utilization. In Section 5.2.1, we show that careful tunning of D can indeed lead to a good tradeoff between these two factors.

5 Evaluation Results

This section presents a systematic performance evaluation of Fair Completion scheduler (FCS) using a diverse sets of workloads, including *Map-heavy* workload, *Reduce-heavy* workload. Furthermore, we conduct stress tests through *Gridmix2* [2]. We compare the performance of FCS to the Hadoop Fair Scheduler (HFS) and Hadoop Capacity Scheduler (HCS). Several versions of Hadoop are available. Particularly, YARN as a successor of Hadoop provides a new framework for task management. However, through code examination and perform evaluation, we have found that YARN adopts the same task schedulers, thus facing the same fairness issues as

Hadoop. In addition, YARN is still not yet ready for large-scale stable execution. Therefore, our evaluation is based on the stable version Hadoop 1.0.4.

5.1 Experimental Environment

Cluster Setup: Experiments are conducted in a cluster of 46 nodes. One node is dedicated as both the NameNode of HDFS and the JobTracker of the Hadoop. Each node is equipped with four 2.67GHz hex-core Intel Xeon X5650 CPUs, 24GB memory, and two 500GB Western Digital SATA hard drives.

Hadoop Setup: We configure 8 map slots and 4 reduce slots per node, based on the number of cores and memory available on each node. We assign 1024MB heap memory to each map and reduce task, respectively. The HDFS block size is set to suggested 128MB [19] to balance the parallelism and performance for MapTasks.

Benchmarks: We employ the well-known *GridMix2* and *Tarazu* benchmarks [6] to demonstrate that FCS is suitable for various types of workloads.

Tarazu benchmarks represent typical jobs in production clusters. Meanwhile, Different benchmarks emphasize different workload characteristics. Map-heavy jobs generate a small amount of intermediate data, thus resulting in lighter ReduceTasks compared to the relatively heavier MapTasks. This group includes *Wordcount*, *TermVector*, *InvertedIndex* and *Kmeans*. On the other hand, Reduce-heavy jobs generate a large amount of intermediate data, thus causing heavy network shuffling and reduce computation at the ReduceTasks. This group includes *TeraSort*, *SelfJoin*, *SequenceCount*, and *RankedInvertedIndex*. it is worth mentioning that we configure the submission of GridMix2 jobs as a Poisson random process with a configurable arriving interval.

Evaluation Metrics: A number of performance metrics used in our presentation are listed as follows.

- ***Average execution time***: This is the plain average of execution time among a group of jobs, reflecting the efficiency of schedulers to a system.
- ***Maximum slowdown***: We refer to *slowdown* as the normalized execution time, which is defined earlier. *Maximum slowdown* is then the biggest slowdown among a group of jobs. This reflects the fairness to jobs of different characteristics.
- ***ReduceTask wait time***: It is defined as the time spent by a ReduceTask in waiting for reduce slots after the same job's MapTasks (i.e. the entire map phase) have all completed. If the ReduceTask gets a slot before that, then the wait time is 0. This aims to reflect the delay experienced by ReduceTasks.
- ***Average preemption times***: This is the average number of preemptions experienced by a group of jobs with similar job sizes. This quantifies the distribution and frequency of preemptions to jobs of different groups that differ in job sizes.

5.2 Evaluating Design Choices of FCS

The design of FCS includes a couple of important design choices such as the threshold parameter that allows task migration to resume a preempted ReduceTask, and the choice of Preemptive ReduceTask instead of killing as the preemption mechanism. In this section, we conduct tests to evaluate these design choices and elaborate their importance.

5.2.1 Opportunistic ReduceTask Migration

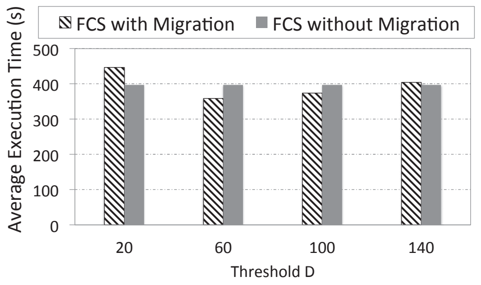


Figure 6: Effectiveness of ReduceTask Migration

As mentioned in section 4, FCS is designed with an opportunistic parameter *D* that controls the tradeoff between keeping ReduceTasks on their original node for data locality and migration ReduceTasks to other available slots for resource utilization. A very large *D* allows a ReduceTask to be delayed many times and become sticky to their original nodes, achieving better data locality for the resumed ReduceTask but at the cost of underutilization of other reduce slots. In contrast, a very small *D* leads to better resource utilization but also incurs more data movement. In this section, we assess the impact of *D* by executing a pool of Gridmix2 jobs. Also, job submission is configured to follow a Poisson random process with an average inter-arrival time of 30 seconds.

In the experiment, we increase *D* from 20 to 140, and compare the performance results of FCS with migration to that of FCS without migration. As shown in Figure 6, FCS with migration can lead to the best average execution time when *D* equals 60, with an improvement of 9.6%. Neither a small D of 20 or a large D of 140 can achieve a good balance between data locality and resource utilization. This experiment confirms that opportunistic task migration as controlled by *D* can lead to good system performance. In the rest of the section, we use 60 as the value for *D*.

5.2.2 Benefits of Preemptive ReduceTask

We investigate the efficiency of FCS when preemption is enabled with either the Preemptive ReduceTask or the killing-based approach. We use three GridMix2 workloads of different numbers of jobs (80 for Workload-

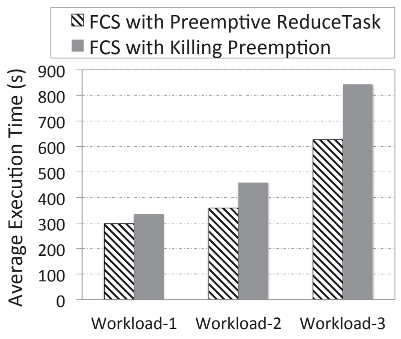


Figure 7: Benefit of Preemptive ReduceTask

1, 130 for Workload-2 and 180 for Workload-3). Figure 7 shows the results. Compared to FCS with killing-based preemption, FCS with Preemptive ReduceTask effectively reduces the average execution time by 11.3%, 21.8% and 25.7% for three workloads, respectively. This demonstrates that FCS performs more efficiently with Preemptive ReduceTask than with the killing approach. In the rest of this paper, we focus on further evaluation of FCS with the Preemptive ReduceTask mechanism.

5.3 Results for Map-heavy Workload

Table 1: Job Composition of Map-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	WordCount	10	1	50
2	TermVector	20	2	40
3	InvertedIndex	50	4	30
4	TermVector	100	8	20
5	Kmeans	500	10	10
6	TermVector	1000	20	8
7	Kmeans	5000	20	6
8	InvertedIndex	10000	60	4
9	TermVector	15000	120	2
10	InvertedIndex	20000	180	1
			Total Jobs	171

Table 2: Performance of Map-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	247	359	1061

We now present the evaluation results on Map-heavy workload. The workload composition is shown in Table 1, featuring two basic characteristics. First, as shown in empirical trace studies [9, 12], realistic workloads exhibit a heavy-tailed distribution for job sizes, accordingly the number of MapTasks among jobs. Second, to capture the effect that jobs arrive to the MapReduce cluster according to a random process, i.e., their arrival interval follows a Poisson random process with an average inter-arrival time of 30 seconds. For ease of presentation, we sort the jobs according to their input sizes and the requested number of tasks, then divide them into 10 different groups of increasing sizes. This categorization helps the interpretation of scheduling effects on jobs of different sizes.

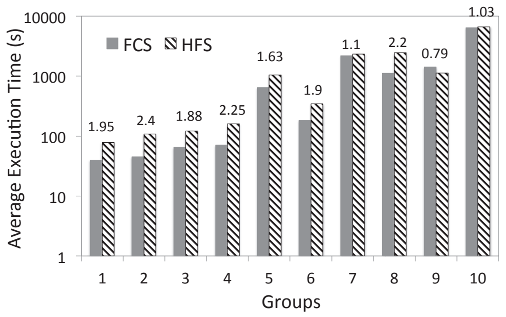


Figure 8: Average Execution Times of Jobs in Different Groups of Map-heavy Workload

Table 2 shows the average execution time for all jobs in Map-heavy workload with different schedulers. Both FCS and HFS significantly outperform HCS, which groups jobs into a small number of job queues, within each queue, HCS adopts FIFO scheduling policy that is known to bias against small jobs and cause long average execution times. Thus we focus on the comparisons between FCS with HFS in the rest performance tests on Map-heavy workload. Overall, FCS speeds up the average execution time by 31% compared to HFS.

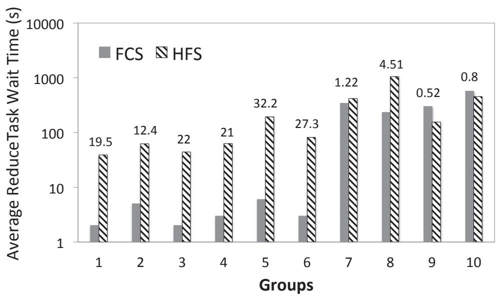


Figure 9: Average ReduceTask Wait Times of Jobs in Different Groups of Map-heavy Workload.

To shed light on how FCS treats jobs of different sizes, we examine the average execution times for the 10 different job groups inside workload. Figure 8 shows that FCS effectively reduces the average execution time for the first 8 groups compared to HFS, achieving up to 2.4× speedup for jobs in group 2. Only jobs in Group 9 are negatively affected by FCS, at an average ratio of 0.79. Such performance results match the design goal of FCS, i.e., trading long running large jobs for fast completion of small jobs.

FCS improves system performance by mitigating the starvation of small jobs. It prioritizes jobs whose shuffle phases are about to complete, thus reducing the ReduceTask wait times. Figure 9 shows the average ReduceTask wait time for all jobs in 10 groups. As we can see, the average wait time is dramatically cut down for the first eight groups by as much as 32.2× for group 5. Only for the last two groups, the wait times are stretched slightly, indicating that FCS yields reduce slots for the small jobs.

To further obtain insights on how FCS has triggered preemptions to different jobs, we record the preemptions experienced by all ReduceTasks. Figure 10 shows the distribution of preemptions to different groups of jobs in the workload. As shown in the figure, no preemptions have happened to Groups 1-4. Groups 5-10 have experienced a small number of preemptions. This demonstrates that FCS can be effective in delivering fair and fast completion without imposing excessive preemptions.

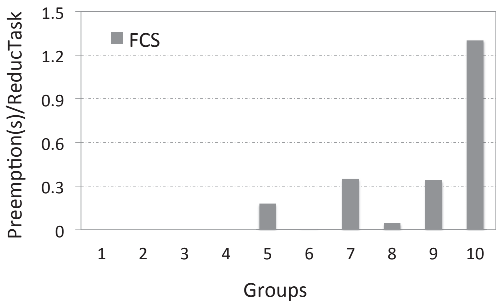


Figure 10: Preemption Frequency

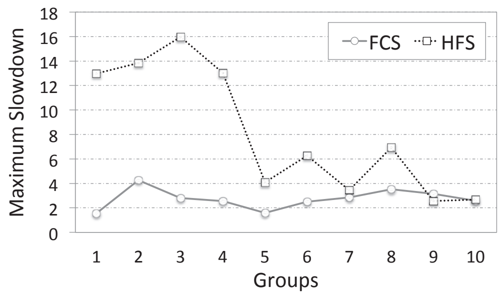


Figure 11: Fairness of Map-heavy Workload

We have measured the maximum slowdown of all jobs to evaluate the fairness of schedulers to different jobs. As shown in Figure 11, FCS efficiently improves the fairness by up to 66.7%, compared to HFS, and achieves nearly uniform maximum slowdown across 10 groups. In contrast, HFS causes serious unfairness to small jobs. In the worse case, a job in Group 3 is slowed down by as much as 16 times.

Taken together, these results confirm the benefits we expect from the design of FCS. They adequately demonstrate the strengths of FCS for Map-heavy workload.

5.4 Results for Reduce-heavy Workload

Map-heavy workload represents jobs that generate small amount of intermediate data. In this section, we continue our evaluation with Reduce-heavy workload, in which jobs generate a large amount of intermediate data, resulting in long running ReduceTasks. The ratio of intermediate data size to input size of those jobs is from 1 : 1 to 3 : 1. The job composition in the workload is listed in Table 3. We adopt the same distributions for job sizes and their arrival times as described in section 5.3.

We conduct the same set of experiments for Reduce-heavy workload as done for the Map-heavy workload to demonstrate that FCS can schedule different workloads effectively. Many results exhibit similar performance to those in Map-heavy workload. Thus, for succinctness, we avoid redundant description, omit some figures, and only highlight the differences. Table 4 shows the overall performance under three schedulers. FCS speeds up the average execution time of the workload by 28% when compared to the HFS, and HCS still performs worse than the other two.

Figure 12 illustrates that FCS speeds up the average execution times of 10 different groups in the workload. This differs from the Map-heavy workload, in which 8 out of 10 groups achieves obvious acceleration. In addition, we observe that FCS improves the completion rate not only for small and medium jobs, but effectively

Table 3: Job Composition of Reduce-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	TeraSort	10	2	50
2	SelfJoin	20	4	40
3	SequenceCount	50	8	30
4	TeraSort	100	16	20
5	SelfJoin	500	32	10
6	RankInvertedIdx	1000	64	8
7	TeraSort	5000	128	6
8	SequenceCount	10000	256	4
9	TeraSort	15000	512	2
10	SequenceCount	20000	1024	1
			Total Jobs	171

Table 4: Performance of Reduce-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	978	1364	8829

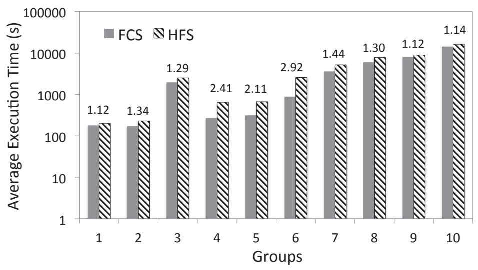


Figure 12: Average Execution Times of Jobs in Different Groups of Reduce-heavy Workload

for the large jobs in the last three groups as well. This is because in Reduce-heavy workload, map phases of large jobs run much longer to generate intermediate data than their counterparts in Map-heavy workload. When small jobs arrive, they preempt long running ReduceTasks from jobs that have not progressed much in the reduce phase. As a result, such preemptions impose little performance impact on the execution of these ReduceTasks in large jobs, because those long running ReduceTasks periodically enter into the idle mode to wait for the availability of intermediate data. On the contrary, these preemptions are greatly beneficial to small jobs that can efficiently utilize the reduce slots to accelerate their execution. Moreover, as small jobs quickly leave the cluster, resource contention is gradually ameliorated. Therefore, long running large jobs obtain resources during the reduce phases and achieve faster job completion.

Similar to the Map-heavy workload, significantly shortened ReduceTask wait time contributes to the fast job completion of Reduce-heavy workload. Figure 13 compares the average ReduceTask wait times between FCS and HFS. For Groups 9 and 10, FCS leads to a slightly longer delay, up to 15%. For Groups 3 and 8, FCS and HFS are comparable. Group 1 has zero wait time in both cases. FCS drastically reduces the wait time down to 0 for Groups 2 and 4, and effectively cuts down the average ReduceTask wait time for Groups 5,6 and 7, ranging from 9.8× to 84.5×. Interestingly, even FCS delays the launch of long running ReduceTasks in Group 9 and 10, their job execution is not affected. In such scenario, more intermediate data is buffered. Once launched, ReduceTasks spend more of their execution on data fetching. Faster completion of all jobs in all groups directly leads to better fairness. Figure 14 shows that FCS efficiently improves the fairness by 35.2% on average when compared to the HFS for the Reduce-heavy workload. Note that FCS still maintains low preemption frequency for different groups of jobs, in particular for large jobs. Because it bears strong resemblance to that of Map-heavy workload, we omit the preemption frequency result here.

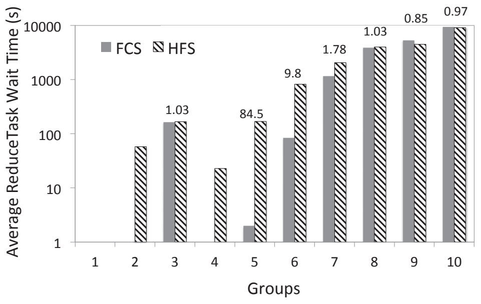


Figure 13: Average ReduceTask Wait Times of Jobs in Different Groups of Reduce-heavy Workload

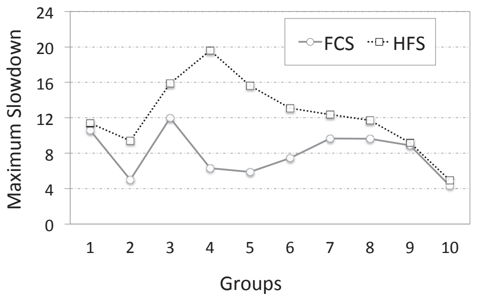


Figure 14: Fairness of Reduce-heavy Workload

5.5 Scalability

The workloads submitted to a production cluster vary substantially over different periods of time. Thus, the capability of efficiently scheduling a large number of randomly arriving jobs is critical for Hadoop schedulers, especially when the system is heavily loaded. To investigate the scalability of FCS, we employ Gridmix to assess the performance of Hadoop when the system is under stress. We vary the number of GridMix jobs from 60 to 300 and maintain the same distribution of job size throughout different tests.

The experimental results are shown in Figure 15. Compared to HFS, FCS consistently reduces the average execution times across different experiments. On average, FCS reduces the average execution time by 39.7%. More importantly, FCS shows stable performance improvement when the number of jobs in the workload increases. The improvement ratio rises from 10% to 28% when the number of jobs increases from 60 to 300. In the workload with 60 jobs, small jobs are dominant with very few large jobs arriving very late. In such a scenario, the demands for reduce slots from small small jobs can be satisfied in time, leading to shortened ReduceTasks waiting time. As a result, it leaves less optimization spaces for FCS to improve. Furthermore, during the tests, when the number of jobs increases, no noticeable scheduling overhead in terms of CPU utilization is observed in the JobTracker.

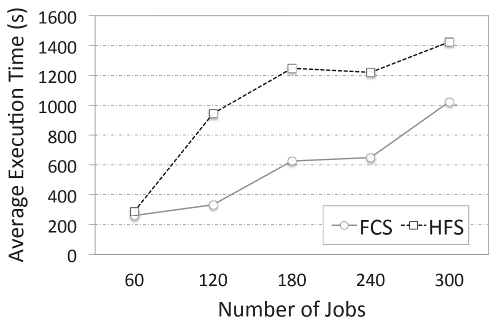


Figure 15: Scalability Evaluation with GridMix2

6 Related Work

Many MapReduce schedulers have been proposed over the past few years trying to maximize the resource utilization in the shared MapReduce clusters. Zaharia *et al.* introduced delay scheduling [19] that speculatively postpones the scheduling of the head-of-line tasks and ameliorate the locality degradation in the default Hadoop Fair scheduler [4]. In addition, Zaharia also proposed Longest Approximate Time to End (LATE) [20] scheduling policy to mitigate the deficiency of Hadoop scheduler in coping with the heterogeneity across virtual machines in a cloud environment. But neither of these two scheduling policies supports task preemption for jobs in the same pool, thus unable to correct the monopolizing behavior of long-running ReduceTasks.

Mantri [7] was designed to mitigate the impact of outliers in MapReduce cluster, it monitors task execution with real-time remaining work estimation, and accordingly take measures such as restarting outliers, placing tasks with network awareness and conserving valuable work from the tasks. But Mantri does not identify the resource monopolizing issue among large number of concurrent jobs caused by long-running ReduceTasks and does not provide lightweight preemption solution. Ahmad [6] proposed communication-aware placement and scheduling of MapTasks and predictive load-balancing for ReduceTasks as part of Tarazu to reduce the network traffic of Hadoop on heterogeneous clusters. But it also does not address the fairness and monopolization issues. Isard *et al.* [11] introduced the Quincy scheduler, which adopts min-cost flow algorithm to achieve a balance between fairness and data locality for the Dryad. But their use of killing as preemption mechanism can cause significant resource waste.

Verma [16] introduced ARIA to allocate appropriate amount of resources to MapReduce job so that it can meet SLO. Based on ARIA, Zhang *et al.* [21] further studied the estimation of required resources for completing a Pig program to meet SLO. Lama [13] proposed AROMA to automatically determine the system configuration for Hadoop jobs to achieve quality of service goal. FLEX [17] aims to optimize different given scheduling metrics based on a performance model between slots and job execution time. However, none of above four work considers the resource contention issue (reduce slot contention) among continuously incoming jobs in shared MapReduce clusters.

In [8], Ananthanarayanan proposed Amoeba which supports lightweight elastic tasks that can release the slots without losing previous I/O and computation. This bears strong similarity to our preemptive ReduceTask. However, it imposes many constraints such as safe points on task processing so that tasks can be interfered without losing previous work. However no overhead measurement is reported in the article. In addition, no corresponding scheduling policy is designed to leverage the benefits provided by elastic task.

Recently, YARN [5] has been proposed by Yahoo! as the next generation MapReduce. It separates the JobTracker into ResourceManager and ApplicationManager, and removes task slot concept. Instead, it adopts resource container concept that encapsulates the general resources, such as memory, CPU and disk I/O into the schedulable unit (current YARN only supports memory). But our initial evaluation discovers that monopolization behavior of long-running ReduceTasks still exist in such framework as long as schedulers greedily allocate as many resources as permitted to one job. Therefore, our Preemptive ReduceTasks and Fair Completion Scheduler can be very beneficial in the new framework. In future, we plan to incorporate our techniques into the YARN.

7 Conclusion

In this paper, we have revealed that there exists a serious fairness issue for the current MapReduce schedulers due to the lack of a lightweight preemption mechanism for ReduceTasks. Accordingly, we have designed and implemented the Preemptive ReduceTask as a work-conserving preemption mechanism, on top of which we have designed the Fair Completion Scheduler. The introduction of the new preemption mechanism and the novel ReduceTask scheduling policy have solved the fairness issue to small jobs, resulting in improved resource utilization and fast average job completion for all jobs. Our design of Fair Completion Scheduler, compared to the Hadoop Fair Scheduler and Capacity Scheduler, can reduce the average job execution time by up to 39.7% and 88.9%, respectively. Furthermore, the Fair Completion Scheduler improves the fairness among different jobs by up to 66.7%, compared to the Hadoop Fair Scheduler.

Acknowledgments
This work is funded in part by a National Science Foundation award CNS-1059376 and by an Alabama Governor's Innovation Award.

References

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Gridmix2. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[3] Hadoop Capacity Scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.

[4] Hadoop Fair Scheduler. http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html.

[5] Next Generation Hadoop MapReduce. http://hadoop.apache.org/docs/current/index.html.

[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 61–74, New York, NY, USA, 2012. ACM.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceeding OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*, Vancouver, BC, Canada, October, 2010. ACM.

[8] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 24:1–24:7, New York, NY, USA, 2012. ACM.

[9] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive query processing in big data systems: A cross industry study of mapreduce workloads. Technical Report UCB/EECS-2012-37, EECS Department, University of California, Berkeley, Apr 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Sixth Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, Dec. 2004.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[12] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010.

[13] P. Lama and X. Zhou. Aroma: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[15] J. Tan, X. Meng, and L. Zhang. Delay tails in mapreduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, New York, NY, USA, 2012. ACM.

[16] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011.

[17] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. balmin. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[21] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.

QoS-Aware Admission Control in Heterogeneous Datacenters

Christina Delimitrou, Nick Bambos and Christos Kozyrakis
Stanford University
{cdel, bambos, kozyraki}@stanford.edu

Abstract

Large-scale datacenters (DCs) host tens of thousands of diverse applications each day. Apart from determining where to schedule workloads, the cluster manager should also decide when to constrain application admission to prevent system oversubscription. At the same time datacenter users care not only for fast execution time but for low waiting time (fast scheduling) as well. Recent work has addressed the first challenge in the presence of unknown workloads, but not the second one.

We present ARQ, a multi-class admission control protocol that leverages Paragon, a heterogeneity and interference-aware DC scheduler. ARQ divides applications in classes based on the quality of resources they need and queues them separately. This improves utilization and system throughput, while maintaining per-application QoS. To enforce timely scheduling, ARQ diverges workloads to a queue of lower resource quality, if no suitable server becomes available within the time window specified by its QoS. In an oversubscribed scenario with 8,500 applications on 1,000 EC2 servers, ARQ bounds performance degradation to less than 10% for 99% of workloads, while significantly improving utilization.

1. Introduction

An increasing amount of computing is performed in the cloud, primarily due to cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [3]. The operator of a cloud service must schedule the stream of incoming applications on available servers in a *resource-efficient* manner, i.e., achieving fast execution (user's goal) at high resource utilization (operator's goal). This scheduling problem is particularly difficult for several reasons, including diverse application characteristics [3, 19], insufficient workload knowledge, co-scheduled application interference and platform heterogeneity. An additional challenge occurs during periods of adversarial traffic, i.e., intervals with very high load, when the system can become oversubscribed, resulting in poor performance. Most DCs employ some admission control to minimize such effects.

DC users are interested in two performance metrics; how fast the application starts running (*waiting time*) and how fast it completes thereafter (*execution time*). While recent work has shown how to improve execution time in the presence of unknown workloads, varying interference sensitivities and heterogeneous servers [14], it does not solve the "head of line blocking" problem [27]. Additionally, some applications have strict scheduling deadlines, while others can tolerate delays in order to be assigned to preferred servers. In all cases, resource requirements should be taken into account at admission point [8].

We propose *ARQ* (*Admission control with Resource Quality-awareness*), a QoS-aware admission control protocol that builds on Paragon and accounts for the resource quality an application needs to preserve its QoS. Resource quality reflects the additional load a server can support without violating application QoS, given its configuration and the applications it currently hosts. ARQ divides workloads to multiple classes and directs them to different queues. This way demanding workloads do not block easy-to-satisfy applications, as they wait for an appropriate server to become available. On the other hand, since DC applications have strict QoS guarantees, they can only be queued for limited amounts of time, while waiting for an appropriate server. ARQ detects when an application is about to violate its performance requirements and re-directs it to a different queue before the QoS violation occurs. We explore the trade-off between waiting time and quality of resources and solve the corresponding optimization problem to find the optimal switching point.

We evaluate ARQ both in small and large-scale experiments. First, we compare the system without and with ARQ in a local cluster with 40 machines and show the benefits in performance and efficiency. We also evaluate ARQ on a 1000-server cluster on Amazon EC2. For an oversubscribed scenario with 8500 applications, Paragon with ARQ guarantees that 99% of workloads have less than 10% performance degradation, while improving utilization by 46%.

2. Background

2.1. Paragon Overview

Paragon is a heterogeneity and interference-aware DC scheduler [14]. It assigns applications to heterogeneous servers based on the platform they benefit from and the co-scheduled applications that minimize destructive interference to preserve QoS. Paragon has two components, a *classification engine* and a *greedy scheduler*. We briefly describe their operation in the following paragraph.

The first component of Paragon performs fast classification of incoming applications, in terms of the server configuration (SC) they perform better on and the interference they cause and tolerate in various shared resources, such as the processor, cache hierarchy, memory, storage and networking subsystems. The interference profile is obtained through targeted microbenchmarks of tunable intensity that create contention in specific shared resources. These microbenchmarks are called sources of interference (SoIs). The classification engine is built as a recommendation system, similar to Netflix [5] or e-commerce systems and leverages the knowledge the system already has about previously-scheduled applications, keeping profiling overheads low. Then, the greedy scheduler searches for a machine of desired SC, that minimizes destructive interference between existing and new load. Paragon scales to tens of thousands of applications and improves utilization, while maintaining per-application QoS.

2.2. Current Limitations

While Paragon shows that accounting for heterogeneity and interference improves resource efficiency without QoS losses, it does not decide when applications should be admitted and scheduled. Paragon accounts for workload characteristics to decide where to assign a workload, but it does not solve the "head of line blocking" problem that can cause high waiting times. By default, applications are scheduled in a simple FIFO order. This has two shortcomings; first, easy-to-satisfy workloads can get trapped behind demanding applications, e.g., workloads that require exclusive instances of high-end, multi-socket servers to preserve their QoS. Second, in the event of an oversubscribed scenario, i.e., when the required resources are more than the total resources available in the system, Paragon implements an application-agnostic admission control protocol. It queues applications in a single queue until the first server becomes available, and then resumes FIFO-ordered scheduling. This ignores the fact that applications need resources of a certain quality to meet their QoS, and can result in performance degradation.

3. Admission Control

3.1. Overview

Large cloud providers such as Amazon EC2 and Windows Azure, typically deploy some admission control protocol. This prevents machine oversubscription, i.e., the same core servicing more than one applications, resulting in high interference and QoS violations.

We design ARQ, a *QoS-aware admission control protocol* that queues and schedules applications based on the quality of resources they need. This solves two problems; first, applications that demand few, easy-to-satisfy resources are not blocked behind demanding workloads. Second, if no suitable servers are available for a given

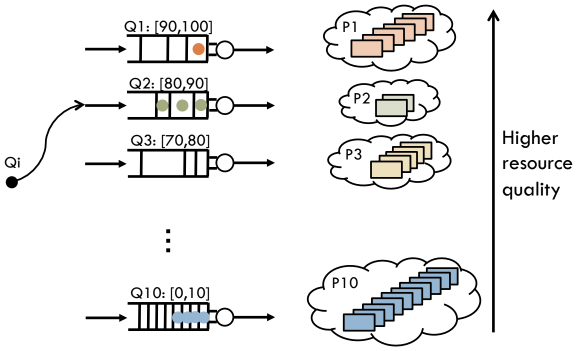


Figure 1: ARQ design. Each queue corresponds to applications with different resource quality requirements.

application, the workload waits for a server of appropriate quality to be freed. Alternatively, the application would be directed to the first free server to avoid queueing delays, with the risk of performance losses.

Resource quality: The resource demands of a workload reflect the load a server should support for the application to meet its QoS. This is a function of the interference the server can tolerate from the new application, and the interference the new workload can tolerate from applications already running on the machine. We use the classification engine in Paragon to derive the per-server tolerated (t_i) and caused (c_i) interference over a set of shared resources. Shared resources include the cache and memory hierarchy, CPU modules, and storage and networking devices. Details on how c_i's and t_i's are obtained can be found in [14]. The interference profile of a server is updated upon initiation or completion of an application's execution. Similarly, upon application arrival, an interference profile is obtained for each new workload. This information guides scheduling decisions by assigning applications to suitable servers. Given the interference profile of a server or application, we define *resource quality* as:

$$Q_i = avg(\sum_i c_i + \sum_i (100 - t_i)) \tag{1}$$

where c_i and t_i are summed over all shared resources for which interference is measured. Conceptually, higher Q_i reflects applications with high demands (high caused and low tolerated interference) that need high-quality system resources. Low Q_i on the other hand, corresponds to workloads that are insensitive to interference, and can satisfy their QoS even when assigned to servers with poor resource quality, e.g., highly-loaded machines, or machines with few cores.

Multi-class admission control: We design ARQ as an admission control protocol with multiple classes of "customers" [1, 6, 17, 20, 21], where customers in this case correspond to applications. The class an application belongs to is determined by its Q_i value. Applications with Q_i values that fall in the same range are assigned to the

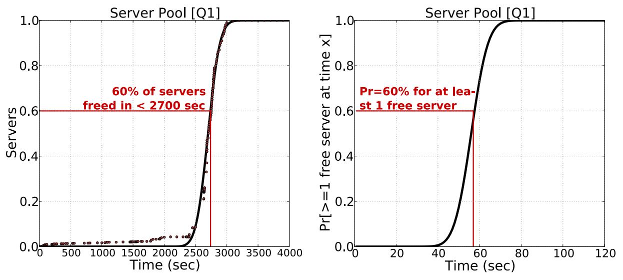


Figure 2: CDF of server busy times and CDF of the probability that there will be at least one free server within a specific time window from an application's arrival.

same class. Q_is range from 0 to 100%. We assume ten classes of applications for now, and justify this selection in the evaluation section (see sensitivity study in Section 5). Fig. 1 shows an overview of ARQ. Each queue corresponds to applications of a specific class. From top to bottom we move from more to less demanding applications. Upon arrival, the cluster manager determines the class an application belongs to and queues it appropriately. Each class has a corresponding server pool of appropriate resource quality. Separating applications based on their resource quality requirements helps ARQ resolve bottlenecks where applications that are sensitive to interference block workloads that are not. On the other hand, applications cannot be queued indefinitely waiting for the perfect server. We address this issue by diverging workloads to queues with better or worse resource qualities.

3.2. Waiting Time versus Resource Quality

Diverging an application to a different queue creates a trade-off between the time an application is waiting in a queue, and the quality of resources it is allocated. We approach this trade-off as an optimization problem.

Queue bypassing: When there is no available server in the pool of a class, queued workloads should be diverged to another queue. There are two possible options for where a workload can be redirected. First, it can be *diverged to a higher queue*. If the queue directly above the queue the workload was originally placed in is empty, the workload is assigned to one of its servers. This hurts utilization, since resources of higher quality than necessary are allocated, but preserves the workload's QoS requirements. In the opposite case the workload is *diverged to a lower queue*. In that case, performance may be degraded, since the application receives resources of lower quality than required. However, the scheme guarantees that in all cases the application will be assigned to a server within the time window dictated by its QoS constraints.

Free-server probability distributions: ARQ needs to know the likelihood that a server of a specific class will become available within the time an application can be queued for, to decide when the workload should be diverged to the next queue. We statistically analyze the server busy time periods for each server pool to obtain these probability distributions. Busy periods are defined as the per-server time intervals from the moment a server is assigned a workload, until that workload completes.

We first use *distribution fitting* to represent the per-pool server busy time in a closed form using known distributions. Fig. 2a shows the CDF of server busy time for the first server pool (highest quality servers) in a 1,000 server experiment. More details on the methodology can be found in Section 4. We show the experimental data (dots) and the closed form representation, derived from distribution fitting. In this case, the data is fitted to a curve resembling a normal distribution. The CDF reflects the fraction of servers that are freed within some time after they have been allocated to an application. For example, 60% of servers in this server pool are freed within 2700 sec from the time an application is scheduled to them.

Using this closed form CDF we easily derive the free-server CDF, which reflects the *probability that within a time interval from an application's arrival, at least one server of the corresponding pool will be available*. Fig. 2b shows the free-server probability CDF for the first server pool. The highlighted point shows that there is a 60% probability that within 56 sec from an application's arrival to that queue, there will be at least one free server in the pool. Free-server CDFs are updated during workload execution to capture changes in application behavior.

Switching between queues: ARQ determines the switching point between queues with the objective to maximize the probability that a server becomes available within a certain window from an application's arrival. For simplicity of explanation we assume that an application's QoS is defined at $0.95x$ of the application's optimal performance. This means that the workload can tolerate at most a 5% performance degradation. Scheduling deadlines or queries-per-second (QPS) can also serve as queueing constraints. Given the free-server CDFs for each server pool, ARQ solves the following optimization problem for application a, switching between queues i and j:

$$max\,\{(S_a - wt_i(t))\cdot Q_i \cdot Pr_i[t], (S_a - wt_j(t))\cdot Q_j \cdot Pr_j[t]\}$$
$$s.t.\,(wt_i(t) + wt_j(t) + P_a) < 0.05\cdot CT_a$$

where $Pr_i[t]$ is the probability that there is a free server in queue i, Q_i is the resource quality of queue i, CT_a is the optimal execution time for application a, P_a is the classification overhead of Paragon, and $S_a = 1.05\cdot CT_a - P_a$ is the available "slack" that can be used for queueing, before the application violates its QoS constraints. ARQ finds the switching time that maximizes the probability that a server of either queue i or j will become available such that the application preserves its QoS guarantees. It also promotes waiting longer for a server of the same class rather than eagerly switching to the next queue ($Q_i > Q_j$).

Server Type	GHz, cores,		L1(KB),	LLC(MB),	mem(GB)	#
Xeon L5609	1.87	2x8	32/32	12	24 DDR3	1
Xeon X5650	2.67	2x12	32/32	12	24 DDR3	2
Xeon X5670	2.93	2x12	32/32	12	48 DDR3	2
Xeon L5640	2.27	2x12	32/32	12	48 DDR3	1
Xeon MP	3.16	4x4	16/16	1	8 DDR2	5
Xeon E5345	2.33	1x4	32/32	8	32 FB-DIMM	8
Xeon E5335	2.00	1x4	32/32	8	16 FB-DIMM	8
Opteron 240	1.80	2x2	64/64	2	4 DDR2	7
Atom 330	1.60	1x2	32/24	1	4 DDR2	5
Atom D510	1.66	1x2	32/24	1	8 DDR2	1

Table 1: Server characteristics of the local cluster. The total core count is 178 for 40 servers of 10 different SCs.

In our analysis we assume batch, single-node applications. In the case of interactive or transactional workloads additional care must be taken to accommodate load changes, e.g., through VM migration. The scheduler detects such changes and adjusts workload placement to preserve QoS. Detection is based on SoI injection and application reclassification.

4. Methodology

Server systems: We evaluated Paragon on a 40-machine local cluster (Table 1) and a 1000-machine cluster with 14 server types on EC2. We used exclusive (reserved) server instances, i.e., there is no interference from external workloads. We also verified that no external scheduling decisions or actions such as auto-scaling or migration are performed during the course of the experiments.

Schedulers: We compared Paragon with ARQ to four schedulers. First, *Paragon* without admission control, second, a *heterogeneity-oblivious* scheme that only accounts for interference but not heterogeneity. Third, an *interference-oblivious* scheme and finally, a scheduler that is both heterogeneity and interference-agnostic, and assigns applications to *least-loaded* machines.

Workloads: We used 29 single-threaded, 22 multi-threaded, 350 multi-programmed and 12 I/O-bound workloads. We use the full SPEC CPU2006 suite and workloads from PARSEC [7], SPLASH-2 [32], BioParallel [18], Minebench [22] and SPECjbb. For multiprogrammed workloads, we use 350 mixes of 4 applications each [26]. The I/O-bound workloads are data mining applications in Hadoop and Matlab. For scenarios with more than 413 applications we replicated these workloads with equal likelihood and randomized their interleaving.

Workload scenarios: For the *small-scale* experiments we examine three workload scenarios. First, we examine a *low-load* scenario with 178 applications, selected randomly from the workload pool, and submitted with 10 sec inter-arrival times. Second, a *high-load* scenario where 178 applications arrive following a Gaussian distribution (μ=10, σ^2=1) that experience significant *phases* during their execution. Finally, we examine a scenario,

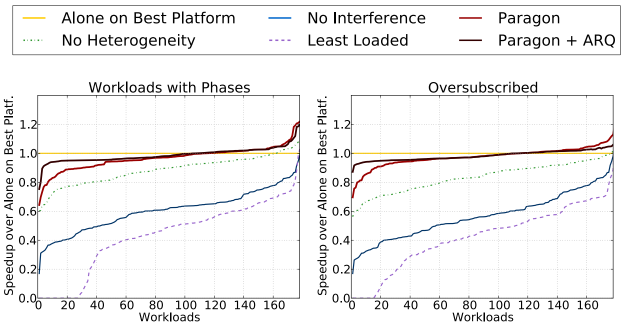


Figure 3: Performance comparison of Paragon and ARQ, across two workload scenarios, against Paragon without admission control, a heterogeneity-oblivious, an interference-oblivious and a least-loaded scheduler.

where 178 applications arrive with 1 sec intervals. This is an *oversubscribed* scenario, since after a few seconds there are not enough resources to execute all applications concurrently. For the *large-scale* experiments on EC2 we examine an oversubscribed scenario where 7,500 workloads arrive with 1 sec intervals and an additional 1,000 applications arrive in burst after the first 3,750 workloads.

5. Evaluation

5.1. Small-scale Experiments

Performance: Fig. 3 shows the performance comparison between the different schedulers for the second and third scenarios in the small-scale cluster. The differences for the low-load scenario where resources are plentiful are small. We focus on the differences between Paragon without and with the use of ARQ. Applications are ordered from worst to best performing. For the scenario with workload phases the applications that preserve their QoS increase from 66% to 91%, and the average performance improves to 99.3%. For the oversubscribed system, while without ARQ only 64% of applications maintain their QoS, with ARQ 88% of workloads preserve their performance requirements. This shows that accounting for resource quality at admission point drains the backlog of queued workloads much faster.

Overheads: ARQ limits waiting time to preserve QoS. Fig. 4 shows the breakdown of execution time for selected applications in the oversubscribed scenario. Time is divided in useful execution time, overheads from training and classification, overheads from the greedy server selection [14] and overheads from queueing. *mcf* and *blackscholes* do not have a bar for the least-loaded (LL) scheduler because they did not complete successfully due to memory exhaustion in the server. In all cases overheads are very low and execution time for most workloads is very close to one (optimal). The overheads from queueing are less than 5% at all times. The cases where queueing is high correspond to workloads that had to be diverged

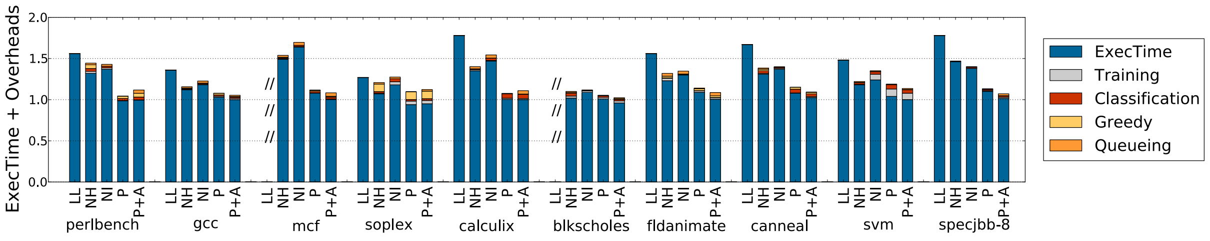


Figure 4: Overheads from classification, queueing and scheduling compared to useful execution time. Overall, the overheads in Paragon with ARQ are less than 5% for most applications.

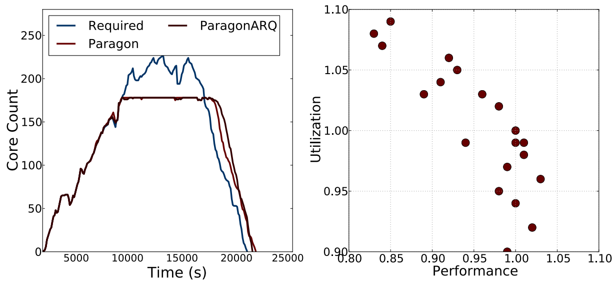


Figure 5: Required versus allocated core count for the oversubscribed scenario in the small-scale system and sensitivity of ARQ to the number of queues. Performance and utilization are normalized to the values for 10 queues.

to queues of lower resource quality, in which case useful execution time is also suboptimal.

Resource allocation: Fig. 5a shows the required versus allocated core count for Paragon with and without ARQ for the oversubscribed scenario. Once the system enters the oversubscribed phase ([9000-17000]sec), Paragon without ARQ allocates all available cores and then queues applications, while Paragon with ARQ will only dispatch applications if an appropriate server is freed. This drains the backlog faster since, even though applications are queued for longer, they run in higher quality platforms.

Server utilization: We also measure server utilization before and after the use of ARQ. We focus on the oversubscribed scenario where ARQ has the highest impact. Paragon without ARQ improves utilization by 47% compared to a LL scheduler. Adding ARQ slightly reduces this improvement since applications are queued instead of being dispatched immediately. Despite this, utilization still improves by 45.5%. This means that the performance benefits of ARQ do not incur an efficiency penalty.

Sensitivity to design parameters: Fig. 5b shows the performance - utilization tradeoff for different numbers of queues. Both metrics are normalized to the values for 10 queues. More queues result in fewer cases of workloads being blocked behind demanding applications, therefore they improve performance, but reduce the number of servers in the corresponding pools, hurting utilization. In contrast, few queues revert to the default scheduler where many applications are scheduled in FIFO order, increasing utilization and hurting performance. 10 queues

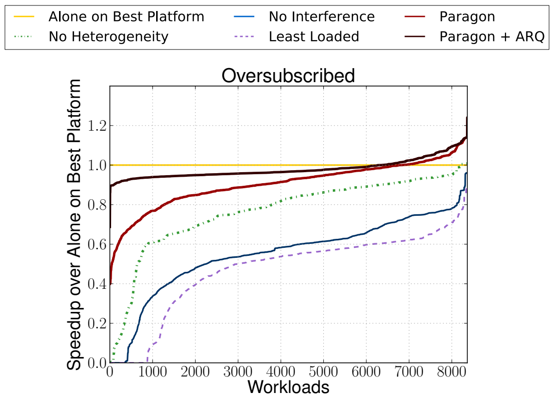


Figure 6: Performance for the different schedulers in the oversubscribed scenario on 1,000 EC2 machines.

achieve both high performance and efficiency.

Large-scale experiments: Fig. 6 compares the performance of the different schedulers for the large-scale scenario. While Paragon without ARQ only preserves QoS for 61% of workloads, introducing admission control increases that fraction to 83%. Additionally, it bounds degradation to less than 10% for 99% of workloads. This shows that the protocol scales well with the number of servers and applications, while maintaining overheads similar to the ones for the small-scale experiments.

6. Conclusions

We have presented ARQ, a QoS-aware admission control protocol for heterogeneous datacenters. ARQ divides applications to classes based on their resource quality requirements and queues them separately in a multi-class network. ARQ is derived from validated queueing models, and it improves system throughput by reducing application waiting time, and diverging workloads to different queues when necessary. In an oversubscribed scenario with 8,500 applications on 1,000 servers, 99% of workloads experience less than 10% degradation compared to 79% of workloads without ARQ.

Acknowledgements

We sincerely thank Christopher Stewart, Daniel Sanchez, and the anonymous reviewers for their feedback on earlier versions of this manuscript. Christina Delimitrou was supported by a Stanford Graduate Fellowship.

References

[1] R. Atar, A. Mandelbaum, M. Reiman. “Scheduling a Multi Class Queue with Many Exponential Server: Asymptotic Optimality in Heavy Traffic”. *In the Annals of Applied Probability, Vol. 14, No. 3, 2004.*

[2] L. Barroso. "Warehouse-Scale Computing: Entering the Teenage Decade". *ISCA Keynote, SJ, June 2011.*

[3] L. A. Barroso, U. Holzle. "The Datacenter as a Computer". *Synthesis Series on Computer Architecture, May 2009.*

[4] L. A. Barroso and U. Holzle. "The Case for Energy-Proportional Computing". *Computer, 40(12):33–37, 2007.*

[5] R. M. Bell. Y. Koren, C. Volinsky. "The BellKor 2008 Solution to the Netflix Prize". Technical report, AT&T Labs, Oct 2007.

[6] D. Bertsimas, D. Gamarnik, J. Tsitsiklis. “Performance of Multiclass Markovian Queueing Networks via Piecewise Linear Lyapunov Functions”. *In Annals of Applied Probability, Vol. 00, No. 0, 1-45, 2001.*

[7] C. Bienia, S. Kumar, et al. ”The PARSEC benchmark suite: Characterization and architectural implications”. *In Proc. of the international conference on Parallel Architectures and Compilation Techniques (PACT), Toronto, 2008.*

[8] J. Carlstrom, R. Rom. “Application-aware Admission Control and Scheduling in Web Servers”. *In Proc. of Infocom, NY, 2002.*

[9] H. Chen. “Fluid Approximations And Stability Of Multiclass Queueing Networks: Work-Conserving Disciplines”. *In Annals of Applied Probability, Vol. 5, No. 3, 1995.*

[10] S. T. Cheng, C. M. Chen, I. R. Chen. “Performance evaluation of an admission control algorithm: dynamic threshold with negotiation”. *In Performance Evaluation 52, 2003.*

[11] J. G. Dai. “On positive Harris recurrence of multiclass queueing networks: A unified approach via fluid limit models”. *In Annals of Applied Probability, 5:49-77, 1995.*

[12] J. G. Dai. “A fluid-limit model criterion for instability of multiclass queueing networks”. *In Annals of Applied Probability, 6:751.757, 1996.*

[13] J. Dean and S. Ghemawat. ”MapReduce: Simplified Data Processing on Large Clusters”. *In Proc. of Symposium on Operating Systems Design and Implementation (OSDI), SF, 2004.*

[14] C. Delimitrou and C. Kozyrakis. “Paragon: QoS-Aware Scheduling in Heterogeneous Datacenters”. *In Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, March 2013.*

[15] Amazon Elastic Compute Cloud-EC2. `http://aws.amazon.com/ec2/`

[16] Google Compute Engine. `http://cloud.google.com/products/compute-engine.html`

[17] J. J. Hasenbein. “Stability, Capacity and Scheduling of Multiclass Queueing Networks”. Ph.D. Thesis. Georgis Institute of Technology, 1998.

[18] A. Jaleel, M. Mattina, B. Jacob. ”Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP - A Case Study of Parallel Bioinformatics Workloads”. *In Proc. of int’l conference on High Performance Computer Architecture (HPCA), TX, 2006.*

[19] C. Kozyrakis, A. Kansal, S. Sankar, K. Vaid. "Server Engineering Insights for Large-Scale Online Services". *In IEEE Micro, vol.30, no.4, July 2010.*

[20] V. G. Kulkarni, N. Gautam. “Admission Control of Multi-Class Traffic with Service Priorities in High-Speed Networks”. *In Journal of Queueing Systems: Theory and Applications archive Vol. 27, No. 1/2, 1997.*

[21] B.L. Miller. “A queueing reward system with several customer classes”. *Management Science, 16(3):234-245, 1969.*

[22] R. Narayanan, B. Ozisikyilmaz, et al. ”MineBench: A Bench- mark Suite for DataMining Workloads”. *In Proc. of the IEEE Int’l Symposium on Workload Characterization (IISWC), SJ, 2006.*

[23] Rackspace. `http://www.rackspace.com/`

[24] A. Rajaraman and J. Ullman. ”Textbook on Mining of Massive Datasets”, 2011.

[25] Amazon EC2: Rightscale. `https://aws.amazon.com/solution-providers/isv/rightscale`

[26] D. Sanchez, C. Kozyrakis. ”Vantage: Scalable and Efficient Fine-Grain Cache Partitioning”. *In Proc. of the International Symposium on Computer Architecture (ISCA), SJ, 2011.*

[27] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. *In Proc. of the European Conference on Systems (EuroSys’13), Prague, Czech Republic, April 2013.*

[28] Tanenbaum, A. S. “Modern Operating Systems” (3rd ed.). *Pearson Education, Inc. p. 156.*

[29] vMotion™. "Migrate VMs with Zero Downtime". `http://www.vmware.com/products/vmotion`

[30] VMWare vSphere. `http://www.vmware.com/products/vsphere/`

[31] Windows Azure. `http://www.windowsazure.com/`

[32] S. Woo, M. Ohara, et al. ”The SPLASH-2 Programs: Characterization and Methodological Considerations”. *In Proc. of the Int’l Symposium on Computer Architecture (ISCA), Italy, 1995.*

[33] Xen Hypervisor 4.0. `http://www.xen.org/`

[34] XenServer. `http://www.citrix.com/English/ps2/products/product.asp?contentID=683148`

Performance Inconsistency in Large Scale Data Processing Clusters

Mingyuan Xia and Nan Zhu
McGill University

Yuxiong He and Sameh Elnikety
Microsoft Research Redmond

Xue Liu
McGill University

Abstract

A large shared computing platform is usually divided into several virtual clusters of fixed sizes, and each virtual cluster is used by a team. A cluster scheduler dynamically allocates physical servers to the virtual clusters depending on their sizes and current job demands. In this paper, we show that current cluster schedulers, which optimize for instantaneous fairness, cause performance inconsistency among the virtual clusters: Virtual clusters with similar loads see very different performance characteristics.

We identify this problem by studying a production trace obtained from a large cluster and performing a simulation study. Our results demonstrate that when using an instantaneous-fairness scheduler, a large VC that contributes more resources during underload periods can not be properly rewarded during its overload periods. These results suggest that not using resource sharing history is the root cause for the performance inconsistency.

1 Introduction

Data-intensive computing is important for a large number of applications, including large-scale data mining, data analytics, and bioinformatics. At the same time, clusters of commodity servers are a major computing platform, powering these large-scale data-intensive applications. Driven by this trend, researchers and practitioners have been developing various cluster computing frameworks to simplify the programming of clusters and to use cluster resources efficiently. Prominent examples include MapReduce [7], Hadoop [4], Dryad [10], and Cosmos [6], among others [19, 15, 17].

A large cluster is normally shared among several teams within an organization, rather than being dedicated to a single team. The benefits of sharing are compelling: First, sharing allows teams to exploit a large number of servers that would be infeasible without sharing. Second, from the system point of view, sharing improves resource utilization by multiplexing the resources among several teams. For example, the web document ranking team in large commercial search engine runs its ranking algorithm (e.g., PageRank) daily for a massive number of crawled documents, running on thousands of servers and lasting for a few hours. Without sharing, the ranking team would need to provision a large dedicated cluster, which will be underutilized.

As a concrete example we consider Cosmos [6], which is a production system that executes jobs similar to those in MapReduce and Hadoop and is used extensively inside Microsoft. A Cosmos cluster can span over 100,000 servers. Organizational units within Microsoft pay for a portion of the cluster, and in return receive a "virtual cluster" (VC). For example, a cluster user (i.e., an organizational unit) pays the cost of 1,000 servers and in return receives a VC of 1,000 servers to run its jobs. Servers in a VC are not dedicated, but are allocated dynamically whenever the VC has jobs. Furthermore, additional idle servers (if available) can be allocated to a busy cluster temporarily. Therefore, although the size of a user VC is only 1,000 servers, its VC can use many more idle servers from other idle VCs.

Sharing brings a key challenge: long term fairness. We want to ensure fairness within a large enough time window among VCs when they compete for resources. Figure 1 shows the performance of 115 VCs in a large Cosmos cluster during one month. Each point represents one VC. The X-axis shows the load, which is equal to the total work (server hours) of the VC in the month divided by its total capacity (number of servers). In other words, load factor = 1 is equivalent to having 100% utilization for the VC during the month. The Y-axis shows the average stretch of jobs in the VC. *Stretch* is response time normalized to job size and VC capacity (as defined later). The figure shows both the merits and challenges of sharing. On the positive side, sharing allows some VCs to use more than their capacity. VCs with load factor

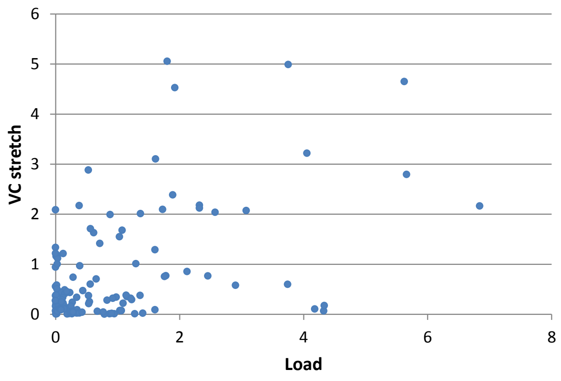


Figure 1: The performance of 115 virtual clusters (VCs) monitored from a large Cosmos system during one month (Sep. 2011).

> 1 benefit from sharing: In particular 10 VCs have load factor > 3, which is equivalent to 300% utilization. Furthermore, overall system utilization is increased; without sharing many clusters would be underutilized. On the negative side, the figure shows a major problem: Some VCs with less than 50% utilization have long stretches (with response times over a few hours), and in contrast some VCs with very high load have short stretches (with very short response times in minutes).

The figure shows long-term unfairness: Two VCs with similar load can have dramatically different responsiveness, and a much more heavily loaded VC may even have much better performance (smaller stretch) than a lightly-loaded VC. This causes several problems to the system operator. Performance inconsistency is the most prominent problem. Teams may even double the size of their VCs with little or no performance improvement. This has direct financial implications since teams are charged for owning servers, and teams paying the same amount of money may have very different performance experiences.

To address these challenges, we perform a trace-driven study based on a production trace of a large Cosmos cluster to reveal the causes of performance inconsistency in real systems. A traditional cluster scheduler [6, 18] uses mainly the current demand and capacity to make scheduling decisions, which we call an instantaneous-fairness scheduler. The well-known MaxMin fairness scheduler [13] and Hadoop fair scheduler [3] are examples of an instantaneous-fairness scheduler. We find that such schedulers do not exploit VC usage and sharing history, and therefore, they do not provide performance consistency among VCs over time (or "long-term fairness").

The contributions of this work are two-fold: (1) We identify the performance inconsistency problem in shared computing clusters. (2) We build a simulator and use a production trace from a large cluster to show how instantaneous-fairness schedulers cause performance inconsistency.

The remainder of the paper is organized as follows: Section 2 elaborates the scheduling model. Section 3 describes our evaluation methodology including workload, simulation design and performance metric. Section 4 uses the simulation results to illustrate performance inconsistency of instantaneous-fairness schedulers and its cause. Section 5 discusses related work and Section 6 discusses the design challenge of the solution. Finally, Section 7 presents our conclusions.

2 Scheduling Model

We explain how users interact with Cosmos and model the system in terms of resource distribution and allocation.

Each Cosmos user (a user here is a team) owns a virtual cluster (VC) that has a *capacity* in terms of the number of servers purchased by the team. A user submits jobs to its VC, and the demand of all jobs in a VC constitutes the VC's *demand*. Instead of allocating a fixed number of servers to VCs, most systems [3, 18] allow VC server *allocation* to fluctuate dynamically: When a VC demands less than its capacity, the VC gets what it demands and the idle servers are allocated to other VCs as additional servers. In return, the VC may receive, during overload, additional servers from under loaded VCs. In our model, we use a_i, c_i and d_i to denote the allocation, capacity and demand of VC_i at time t. We focus on the system scheduler that allocates servers to VCs instead of the VC-level scheduler that schedules jobs within a VC. We assume that the jobs are malleable: the number of servers allocated to a job can be adjusted during the job's execution; Cosmos and MapReduce jobs belong to this category.

3 Experimental Methodology

We evaluate performance consistency using trace-driven simulations with workload from a commercial data center. We use the well-known MaxMin scheduler [13] for instantaneous fairness. Notice that the widely adopted Hadoop Fair scheduler [3] is also an example of a MaxMin scheduler.

3.1 Workload

Cosmos [6] is a large production data-intensive computation platform system similar to MapReduce systems. Cosmos clusters contain tens of thousands of servers for hundreds of users (VCs). We collect a one-month trace (Sep 2011) of a commercial cluster containing about 50,000 servers shared by 115 users. We observe that job distribution does not follow the usual diurnal cycle, as the system serves teams from all over the world. This fact brings more challenges because the workload behavior and size of jobs are more diverse.

To reproduce the diversity of workload behavior, we choose six VCs (two under-utilized, three fully-utilized and one over-utilized) with different load characteristics to assess the performance inconsistency. Figure 2 depicts the daily aggregated load of the six VCs as well as the daily load of each VCs. The figure shows that it is common that one VC is over-demanding while another is under-demanding. Under such circumstances, sharing is a major factor that affects VC performance. Scheduling resources to achieve performance consistency is critical in such a sharing system.

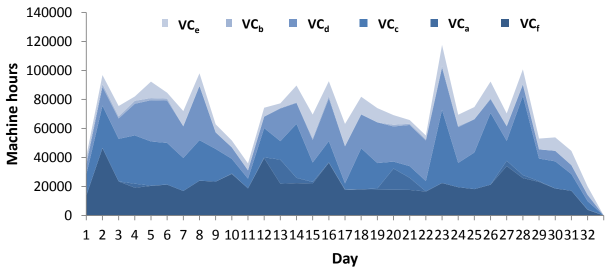


Figure 2: The load fluctuation of six VCs in the one-month trace. The Y-axis is the amount of work of the day. One machine hour denotes the work done by one machine in one hour.

3.2 Simulation design

Trace-driven simulator. We build a trace-driven simulator using desmoj [1], which is a discrete-event library. Our simulator replays a trace from a trace file containing job information, including the submission time, job size, and parallelism. The output of the simulator includes detailed execution information for each job as well as general statistical information such as mean response time.

The total number of machines simulated is 4,000, which is a sum of the capacity of the six VCs plus additional 1,250 machines owned by Cosmos system. Cosmos has additional machines for providing fault tolerance and for running system maintenance jobs; these machines are available to the VCs when they are not running system jobs.

3.3 Performance metric

We measure the performance of each job using the *stretch* metric, which indicates normalized responsiveness of the job. Stretch is defined as the execution time divided by the ideal execution time of the job. To compute the ideal execution time, we divide the job size (in

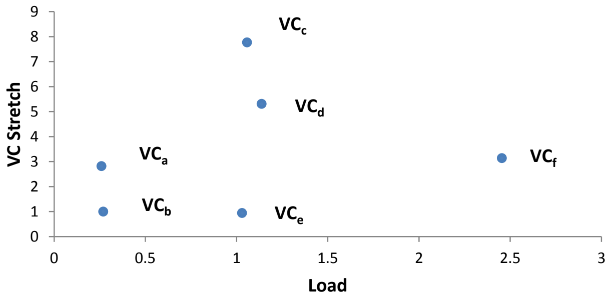


Figure 3: VC stretch with different loads under production workload.

terms of server hours) by its ideal resource allocation — when the job takes the entire VC capacity. We choose ideal resource allocation as the capacity of its VC based on two considerations. On one hand, if idea resource allocation is smaller than the VC capacity, then the VC may require at least two concurrently running jobs to fully use its resources. On the other hand, if it is higher than the VC capacity, the VC will always over-demand its capacity (even with only one running job), which obviates sharing opportunities. Therefore, the definition of job stretch is as follows:

$$\text{job stretch} = \frac{\text{real execution time}}{\text{ideal execution time}} = \frac{\text{real execution time}}{\text{job size}/\text{VC capacity}}.$$

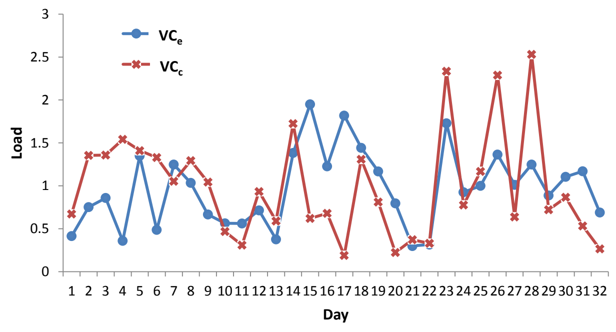
Notice that job stretch is a normalized performance metric and thus overcomes the shortcomings of real-valued metrics such as response time. As jobs have diverse sizes, comparing the response time of jobs from two VCs may not be meaningful. Stretch eliminates this drawback. A larger stretch of a job indicates the job performs worse. In particular, a job with stretch of 1 means that the job is performing the same as the case that the job owns the entire VC by itself. The VC stretch is the mean stretch of all its containing jobs. In later experiments, we distinguish between VC stretch and job stretch.

Stretch can be computed once the job has finished, as job size can be obtained only after job completion.

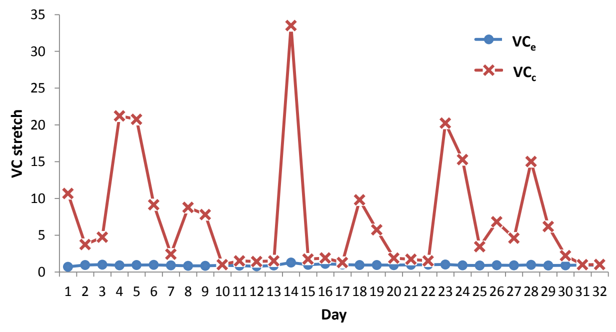
4 Experimental Results

This section illustrates performance inconsistency and its cause based on simulation results driven by Cosmos trace of a commercial data center.

Figure 3 shows the results when simulated with the MaxMin scheduler. We want VCs with similar load to have similar performance, and we call this property performance consistency. However, as shown in the results, three fully-utilized VCs (VC_c, VC_d and VC_e) with different burstiness patterns observe different performance. In the meantime, the over-utilized VC (VC_f) has a better performance than two fully-utilized VCs. These results show that instantaneous-fairness schedulers do not maintain performance consistency among VCs with similar load; They fail to provide long-term fairness for practical workloads. Furthermore, to reveal the cause of this performance inconsistency, we choose two fully-utilized VCs with different performance and examine their load and performance over time.



(a) The load fluctuation.



(b) The VC stretch. Lower is better.

Figure 4: Daily load and performance of two VCs with similar overall load.

Figure 4 inspects the daily load fluctuation and performance for the two VCs (VC_c and VC_e). The two VCs have similar load but different capacities as well as load characteristics. VC_c is a large VC with a capacity of 900 servers while VC_e is smaller with a capacity of 350. Although both VCs are fully utilized ($load = 1$), their loads fluctuate daily, as shown in Figure 4(a). As for the performance, both VCs perform well ($stretch = 1$) during underload days ($load < 1$). But for overload days, VC_e can still perform well while the performance of VC_c is largely degraded, which demonstrates an unfair situation.

The load-performance behavior seen in Figure 4 is closely related to the scheduling algorithm. A traditional instantaneous scheduler, such as the MaxMin Scheduler [13] and the Hadoop Fair Scheduler [3, 18], typically maximizes the minimum allocation for all VCs at a given time point. More specifically, an instantaneous scheduler has the following four properties:

1) When a VC demands less than its capacity, the instantaneous scheduler always fully satisfies the VC demands. So in Figure 4, as long as the daily load of VC_c or VC_e is smaller than 1, its stretch is equal to 1. The free servers from these underloaded VCs will then be assigned to other over-demanding VCs if there are any.

2) When a VC is over-demanding, it competes with other over-demanding VCs for free servers contributed by underloaded VCs. So these VCs may not have a stretch of one. This explains why VC_c's performance degrades for overload days.

3) When competing for additional free servers, smaller VCs have a higher probability to be fully satisfied than larger ones with similar load. This is because when the load is the same, the exceeding demand is proportional to VC capacity. So to maximize the minimum allocation for all VCs, an instantaneous scheduler has to prioritize satisfying less demanding VCs, which makes it harder for large VCs to obtain extra allocation.

4) The scheduling decision is only made at a given time point. So even if a large VC contributes more resources during underload periods, it has to compete equally with other VCs during overload periods. As a result, a bursty large VC may fail to receive enough resource during busy hours regardless how many resources it has contributed earlier.

This case study demonstrates how an instantaneous scheduler casues long-term unfairness. A large VC that contributes more resources during underload periods cannot be properly rewarded during its overload periods. And this situation is caused by the nature of instantaneous fairness, where the sharing history is not considered for scheduling decision.

5 Related Work

The Hadoop Fair Scheduler is widely adopted in multiuser Hadoop clusters [18, 3]. It divides the Hadoop cluster into pools and assigns a pool to each user. The scheduler computes the fair share of each pool according to instantaneous information such as the weight, minimum share and demands of pools, without considering the resource usage history. Hadoop Fair Scheduler is an example of MaxMin Fair Scheduler used in datacenters; other schedulers in this category, including the Hadoop Capacity Scheduler [2], and Quincy [11], consider fairness at the moment of allocation rather than cluster usage history. The Dominant Resource Fair Scheduler [8, 9] schedules multiple types of resources to improve fairness and utilization. When there is only one type of resources, it performs exactly as a MaxMin fair scheduler. Variations of MaxMin scheduler are also used for scheduling shared-memory multiprocessor systems [14, 5]. All of the above prior work do not consider usage history, and therefore, they cannot guarantee performance consistency, which is the focus of this work.

6 Discussion

We show in our experiment that not considering usage history serves as the root cause of performance inconsistency. So here we first present several existing history-based schedulers that can be potentially useful for our scenario. Then we explain the particular requirement of such a scheduler in large-scale data processing systems such as Cosmos. The discussion focuses on designing a practical history-based scheduler for similar systems.

Deficit Round-Robin (DRR) scheduler [16] proposes a technique that allows each flow passing through a network device to have a fair share of network resources. As packet size may differ, simple round robin algorithm may not be fair; DRR uses a deficit counter as a representation of usage history to revise the round robin algorithm to achieve long-term fairness. The Xen credit scheduler [12] applies similar mechanisms to allow multiple virtual machines to fairly share CPU resources.

Both schedulers regulate user's future resource allocation using a counting scheme that measures the previous usage. The counting scheme ensures that a user that overuses its fair share in previous time slot will be charged evenly (or even more) in the future. This guarantees long-term fairness, i.e., over-demanding users will not hurt other users. However, from the view of system operators, promoting overall system utilization is as important as maintaining fairness among users. We argue that using existing strict history-based schedulers will harm the overall utilization to a certain extent. For example, in order to promote overall utilization, the system operators should encourage users to use the system when it is under-loaded. However, by using existing schedulers, over-demanding users will always be penalized in the future regardless of how under-loaded the overall system is. As a result, these schedulers fail to provide incentive for users to use the system during idle periods, which in turn reduces the overall utilization. Thus balancing the system utilization and fairness is an important design challenge for long-term fair schedulers in large-scale data processing systems.

7 Conclusion

A large computing cluster is normally shared among users within an organization to have high system utilization and to offer more computing resources. However, sharing comes with an important fairness problem, resulting in performance inconsistency among users. We identify this problem by conducting a simulation study using a production trace from a large cluster. The results show that traditional cluster schedulers that optimize for instantaneous fairness cannot guarantee performance consistency in the long term. The main reason for this is that instantaneous-fairness schedulers do not consider the sharing history of users. As a result, users with large and bursty workloads do not gain credits for contributing to the system during idle periods. In contrast, they may observe bad performance during busy periods. Our study demonstrates that instantaneous-fairness schedulers may incur performance inconsistency in long run so sharing history should be utilized to provide a better scheduling decision.

Acknowledgement

The authors thank Omer Reingold and Moshe Babaioff (from Microsoft Research) for valuable discussions and feedback.

References

[1] A framework for discrete-event modelling and simulation. `http://desmoj.sourceforge.net/home.html`.

[2] Hadoop capacity scheduler. `http://hadoop.apache.org/docs/stable/capacity_scheduler.html`.

[3] Hadoop fair scheduler. `http://hadoop.apache.org/docs/stable/fair_scheduler.html`.

[4] The hadoop project. `http://hadoop.apache.org/`.

[5] AGRAWAL, K., HE, Y., HSU, W. J., AND LEISERSON, C. E. Adaptive scheduling with parallelism feedback. In *IPDPS* (2007).

[6] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (jan 2008), 107–113.

[8] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI* (2011).

[9] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[10] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys* (2007).

[11] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP 2009* (2009).

[12] KANG, H., CHEN, Y., WONG, J. L., SION, R., AND WU, J. Enhancement of xen s scheduler for mapreduce workloads. In *HPDC* (2011).

[13] KESHAV, S. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] MCCANN, C., VASWANI, R., AND ZAHORJAN, J. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst. 11*, 2 (1993), 146–178.

[15] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[16] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transaction on Networking 4*, 3 (1996), 375–385.

[17] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., LFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008).

[18] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Job scheduling for multi-user mapreduce clusters. Tech. rep., Berkeley, 2009.

[19] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2009).

Temperature Aware Workload Management in Geo-distributed Datacenters

Hong Xu, Chen Feng, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

For geo-distributed datacenters, lately a workload management approach that routes user requests to locations with cheaper and cleaner electricity has been shown promising in reducing the energy cost. We consider two key aspects that have not been explored before. First, the energy-gobbling cooling systems are often modeled with a location-independent efficiency factor. Yet, through empirical studies, we find that their actual energy efficiency depends directly on the ambient temperature, which exhibits a significant degree of geographical diversity. Temperature diversity can be used to reduce the overall cooling energy overhead. Second, datacenters run not only interactive workloads driven by user requests, but also delay tolerant batch workloads at the back-end. The elastic nature of batch workloads can be exploited to further reduce the energy consumption.

In this paper, we propose to make workload management for geo-distributed datacenters *temperature aware*. We formulate the problem as a joint optimization of request routing for interactive workloads and capacity allocation for batch workloads. We develop a distributed algorithm based on an m-block *alternating direction method of multipliers* (ADMM) algorithm that extends the classical 2-block algorithm. We prove the convergence of our algorithm under general assumptions. Through trace-driven simulations with real-world electricity prices, historical temperature data, and an empirical cooling efficiency model, we find that our approach is consistently capable of delivering a 15%–20% cooling energy reduction, and a 5%–20% overall cost reduction for geo-distributed clouds.

1. INTRODUCTION

Geo-distributed datacenters operated by organizations such as Google and Amazon are the powerhouses behind many Internet-scale services. They are deployed across the Internet to provide better latency and redundancy. These datacenters run hundreds of thousands of servers, consume megawatts of power with massive carbon footprint, and incur electricity bills of millions of dollars [17, 34]. Thus, the topic of reducing their energy consumption and cost has received significant attention [7, 11–13, 15, 17, 19, 26–29, 34, 35, 40].

Energy consumption of individual datacenters can be reduced with more energy efficient hardware and integrated thermal management [7, 11, 15, 28, 40]. Recently, important progress has been made on a new *workload management* approach that instead focuses on the overall energy cost of geo-distributed datacenters. It exploits the geographical diversity of electricity prices by optimizing the *request routing* algorithm to route user requests to locations with cheaper and cleaner electricity [12, 17, 18, 26, 27, 29, 34, 35].

In this paper, we consider two key aspects of geo-distributed datacenters that have not been explored in the literature.

First, cooling systems, which consume 30% to 50% of the total energy [33, 40], are often modeled with a constant and location-independent energy efficiency factor in existing efforts. This tends to be an over-simplification in reality. Through our study of a state-of-the-art production cooling system (Sec. 2), we find that temperature has direct and profound impact on cooling energy efficiency. This is especially true with *outside air cooling* technology, which has seen increasing adoption in mission-critical datacenters [1–3]. As we will show, its partial PUE (power usage effectiveness), defined as the sum of server power and cooling overhead divided by server power, varies from 1.30 to 1.05 when temperature drops from 35 °C (90 °F) to -3.9 °C (25 °F).

Through an extensive empirical analysis of daily and hourly climate data for 13 Google datacenters, we further find that temperature varies significantly across both time and location, which is intuitive to understand. These observations suggest that datacenters at different locations have distinct and time-varying cooling energy efficiency. This establishes a strong case for making workload management *temperature aware*, where such temperature diversity can be used along with price diversity in making request routing decisions to reduce the overall cooling energy overhead for geo-distributed datacenters.

Second, energy consumption comes not only from interactive workloads driven by user requests, but also from delay tolerant batch workloads, such as indexing and data mining jobs, that run at the back-end. Existing efforts focus mainly on request routing to minimize the energy cost of interactive workloads, which is only a part of the entire picture. Such a mixed nature of datacenter workloads, verified by measurement studies [36], provides more opportunities to utilize the cost diversity of energy. The key observation is that batch workloads are elastic to resource allocations, whereas interactive workloads are highly sensitive to latency and have more profound impact on revenue [25]. Thus at times when one location is comparatively cost efficient (in terms of dollar per unit energy), we can increase the capacity for interactive workloads by reducing the resources for batch jobs. More requests can then be routed to and processed at this location, and the cost saving can be more substantial. We thus advocate a holistic workload management approach, where *capacity allocation* between interactive and batch workloads is dynamically optimized with request routing. Dynamic capacity allocation is also technically feasible because jobs run on highly scalable systems such as MapReduce.

Towards temperature aware workload management, we propose a general framework to capture the important tradeoffs involved (Sec. 3). We model both energy cost and utility loss, which correspond to performance-related revenue reduction. We develop an empirical cooling efficiency model based on a production system. The problem is formulated as a joint optimization of request routing and capacity allocation. The technical challenge is then to develop a distributed algorithm to solve the large-scale optimization with tens of millions of variables for a production geo-distributed cloud. Dual decomposition with subgradient methods are often used to develop distributed optimization algorithms. However they require delicate adjustments of step sizes that make convergence difficult to achieve for large-scale problems. The method of multipliers [22] achieves fast convergence, at the cost of tight coupling among variables.

We rely on the *alternating direction method of multipliers* (ADMM), a simple yet powerful algorithm that blends the advantages of the two approaches. ADMM recently has found practical use in many large-scale distributed convex optimization problems in machine learning and data mining [10]. It works for problems whose objective and variables can be divided into *two* disjoint parts. It alternatively optimizes part of the objective with one block of variables to iteratively reach the optimum. Our formulation has three blocks of variables, yet little is known about the convergence of m-block ($m \geq 3$) ADMM algorithms, with two exceptions [20, 23] very recently. [20] establishes the convergence of m-block ADMM for strongly convex objective functions, but not linear convergence; [23] shows the linear convergence of m-block ADMM under the assumption that the relation matrix is full column rank, which is, however, not the case in our formation. This motivates us to refine the framework in [23] so that it can be applied to our setup.

In particular, in Sec. 4 we show that by replacing the full-rank assumption with some mild assumptions on the objective functions, we are not only able to obtain the same convergence and rate of convergence result, but also to simplify the proof of [23]. The m-block ADMM algorithm is general and can be applied in other problem domains. For our case, we further develop a distributed algorithm in Sec. 5, which is amenable to a parallel implementation in datacenters.

We conduct extensive trace-driven simulations with real-world electricity prices, historical temperature data, and an empirical cooling efficiency model to realistically assess the potential of our approach (Sec. 6). We find that temperature aware workload management is consistently able to deliver a 15%–20% cooling energy reduction and a 5%–20% overall cost reduction for geo-distributed datacenters. The distributed ADMM algorithm converges quickly within 70 iterations, while a dual decomposition approach with subgradient methods fails to converge within 200 iterations. We thus believe our algorithm is practical for large-scale real-world problems.

2. BACKGROUND AND MOTIVATION

Before we make a case for temperature aware workload management, it is necessary to introduce some background of datacenter cooling, and empirically assess the geographical diversity of temperature.

2.1 Datacenter Cooling

Datacenter cooling is provided by the computer room air conditioners (CRACs) placed on the raised floor of the facility. Hot air exhausted from server racks travels through a cooling coil in the CRACs. Heat is often extracted by chilled water in the cooling coil, and the returned hot water is cooled through mechanical refrigeration cycles in an outside chiller plant continuously. The compressor of a chiller consumes a massive amount of energy, and accounts for the majority of the overall cooling cost [40]. The result is an energy-gobbling cooling system that typically consumes a significant portion ($\sim$30%) of the total datacenter power [40].

2.2 Outside Air Cooling

To improve energy efficiency, various so-called free cooling technologies that operate without mechanical chillers have recently been adopted. In this paper, we focus on a more economically viable technology called *outside air cooling*. It uses an air-side economizer to direct cold outside air into the datacenter to cool down servers. The hot exhaust air is simply rejected out instead of being cooled and recirculated. The advantage of outside air cooling can be significant: Intel ran a 10-month experiment using 900 blade servers, and reported that 67% of the cooling energy can be saved with only slightly increased hardware failure rates [24]. Companies like Google [1], Facebook [2], and HP [3] have been operating their datacenters with up to 100% outside air cooling, which brings million dollars of savings annually.

The energy efficiency of outside air cooling heavily depends on ambient temperature among other factors. When temperature is lower, less air is needed for heat exchange, and the air handler fan speed can be reduced to save energy. Thus, a CRAC with an air-side economizer usually operates in three modes. When ambient temperature is high, outside air cooling cannot be used, and the CRAC falls back to mechanical cooling with chillers. When temperature falls below a certain threshold, outside air cooling is utilized to provide partial or entire cooling capacity. When temperature is too low, outside air is mixed with exhaust air to maintain a suitable supply air temperature. In this mode, CRAC energy efficiency cannot be further improved since fans need to operate at a minimum speed to maintain airflow. Table 1 shows the empirical COP[1] and partial PUE (pPUE)[2] data of a state-of-the-art CRAC with an air-side economizer. Clearly, as the outdoor temperature drops, the CRAC switches the operating mode to use more outside air cooling. As a result the COP improves six-fold from 3.3 to 19.5, and the pPUE decreases dramatically from 1.30 to 1.05. Due to the sheer amount of energy a datacenter draws, the numbers imply huge monetary savings for the energy bill.

Outdoor ambient	Cooling mode	COP	pPUE
35°C(90°F)	Mechanical	3.3	1.30
21.1°C(70°F)	Mechanical	4.7	1.21
15.6°C(60°F)	Mixed	5.9	1.17
10°C(50°F)	Outside air	10.4	1.1
-3.9°C(25°F)	Outside air	19.5	1.05

Table 1: Efficiency of Emerson's DSE™ cooling system with an EconoPhase air-side economizer [14]. Return air is set at 29.4°C(85°F).

With the increasing use of outside air cooling, this finding motivates our proposal to make workload management temperature aware. Intuitively, datacenters at colder and thus more energy efficient locations should be better utilized to reduce the overall energy consumption and cost simultaneously. Our idea also applies to datacenters using mechanical cooling, because contrary to previous work's assumption [28], as shown in Table 1, the chiller energy efficiency also depends on outside temperature, albeit milder.

2.3 An Empirical Climate Study

Our idea hinges upon a key assumption: Temperatures are diverse and not well correlated at different locations. In this section, we make our case concrete by supporting it with an empirical analysis of historical climate data.

We use Google's datacenter locations for our study, as they represent a global production infrastructure and the location information is publicly available [4]. Google has 6 datacenters in the U.S., 1 in South America, 3 in Europe, and 3 in Asia. We acquire historical temperature data from various data repositories of the National Climate Data Center [6] for all 13 locations, covering the entire one-year period of 2011.

It is useful to first understand the climate profiles at individual locations. Figure 1 plots the daily average temperatures for three select locations in North America, Europe,

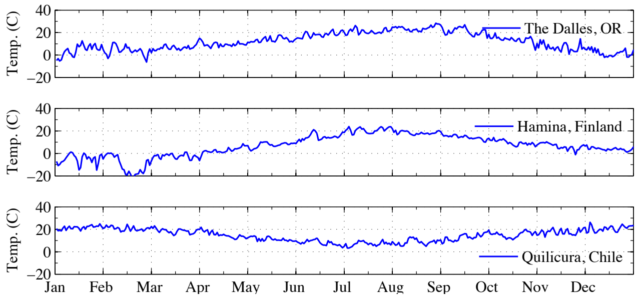


Figure 1: Daily average temperature at three Google datacenter locations. Data from the Global Daily Weather Data of the National Climate Data Center (NCDC) [6]. Time is in UTC.

and South America, respectively. Geographical diversity exists despite the clear seasonal pattern shared among all locations. For example, Finland appears to be especially favorable for cooling during winter months. Diversity is more salient for locations in different hemispheres (e.g. Chile). We also observe a significant amount of day-to-day volatility, suggesting that the availability and capability of outside air cooling constantly varies across regions, and there is no single location that is always cooling efficient.

We then examine short-term temperature volatility. As shown in Figure 2, the hourly variations are more dramatic and highly correlated with time-of-day, which is intuitive to understand. Further, the highs and lows do not occur at the same time for different regions due to time differences.

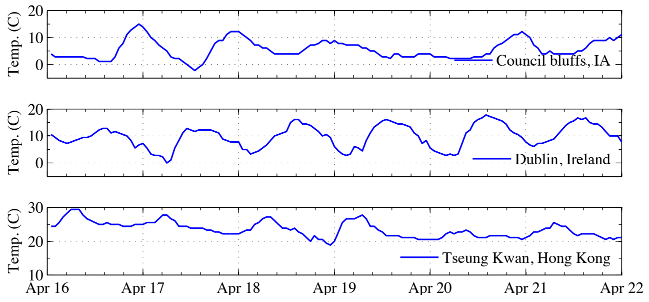


Figure 2: Hourly temperature variations at three Google datacenter locations. Data from the Hourly Global Surface Data of NCDC [6]. Time is in UTC.

Our approach would fail if hourly temperatures are well correlated at different locations. However, we find that this is not the case for datacenters that are usually far apart from each other. The pairwise temperature correlation coefficients for all 13 locations are mostly in between 0.6 and -0.6. Due to space limit, details are omitted and can be found in Sec. 2.3 of our technical report [39].

The analysis above reveals that for globally deployed datacenters, local temperature at individual locations exhibits both time and geographical diversity. Therefore, a carefully designed workload management scheme is critically needed,

[1]COP, coefficient of performance, is defined for a cooling device as the ratio between cooling capacity and power.

[2]pPUE is defined as the sum of cooling capacity and cooling power divided by cooling capacity. Nearly all the power delivered to servers translates to heat, which matches the CRAC cooling capacity.

in order to dynamically adjust datacenter operations to the ambient conditions, and to save the overall energy costs.

3. MODEL

In this section, we introduce our model first and then formulate the temperature aware workload management problem of joint request routing and capacity allocation.

3.1 System Model

We consider a discrete time model where the length of a time slot matches the time scale at which request routing and capacity allocation decisions are made, *e.g.*, hourly. The joint optimization is periodically solved at each time slot. We therefore focus only on a single time slot.

We consider a provider that runs a set of datacenters $\mathcal{J}$ in distinct geographical regions. Each datacenter $j \in \mathcal{J}$ has a fixed capacity C_j in terms of the number of servers. To model datacenter operating costs, we consider both the *energy cost* and *utility loss* of request routing and capacity allocation, which are detailed below.

3.2 Energy Cost and Cooling Efficiency

We focus on servers and cooling system in our energy cost model. Other energy consumers, such as network switches, power distribution systems, etc., have constant power draw independent of workloads [15] and are not relevant.

For servers, we adopt the empirical model from [15] that calculates the individual server power consumption as an affine function of CPU utilization, $P_{\text{idle}} + (P_{\text{peak}} - P_{\text{idle}})\,u$. P_{idle} is the server power when idle, P_{peak} is the server power when fully utilized, and u is the CPU load. This model is especially accurate for calculating the aggregated power of a large number of servers [15]. Thus, assuming workloads are perfectly dispatched and servers have a uniform utilization as a result, the server power of datacenter j can be modeled as $C_j P_{\text{idle}} + (P_{\text{peak}} - P_{\text{idle}})\,W_j$, where W denotes the total workload in terms of the number of servers required.

For the cooling system, we take an empirical approach based on production CRACs to model its energy consumption. We choose not to rely on simplifying models for the individual components of a CRAC and their interactions [40], because of the difficulty involved in and the inaccuracy resulted from the process, especially for hybrid CRACs with both outside air and mechanical cooling. Therefore, we study CRACs as a black box, with outside temperature as the input, and its overall energy efficiency as the output.

Specifically, we use partial PUE (pPUE) to measure the CRAC energy efficiency. As in Sec. 2.2, pPUE is defined as

$$\text{pPUE} = \frac{\text{Server power} + \text{Cooling power}}{\text{Server power}}.$$

A smaller value indicates a more energy efficient system. We apply regression techniques to the empirical pPUE data of the Emerson CRAC [14] introduced in Table 1. We find that the best fitting model describes pPUE as a quadratic function of the outside temperature as plotted below.

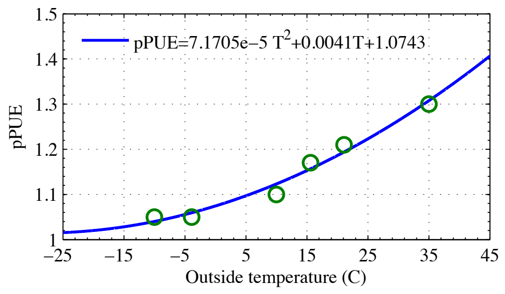


Figure 3: Model fitting of pPUE as a function of the outside temperature T for Emerson's DSE™ CRAC [14]. Small circles denote empirical data points.

The model can be calibrated given more data from measurements. For the purpose of this paper, our approach yields a tractable model that captures the overall CRAC efficiency for the entire spectrum of its operating modes. Our model is also useful for future studies on datacenter cooling energy.

Given the outside temperature T_j, the total datacenter energy as a function of the workload W_j can be expressed as

$$E_j(W_j) = (C_j P_{\text{idle}} + (P_{\text{peak}} - P_{\text{idle}})\,W_j) \cdot \text{pPUE}(T_j). \quad (1)$$

Here we implicitly assume that T_j is known a priori and do not include it as the function variable. This is valid since short-term weather forecast is fairly accurate and accessible.

A datacenter's electricity price is denoted as P_j. The price may additionally incorporate the environmental cost of generating electricity [17], which we do not consider here. In reality, electricity can be purchased from local day-ahead or hour-ahead forward markets at a pre-determined price [34]. Thus, we assume that P_j is known a priori and remains fixed for the duration of a time slot. The total energy cost, including server and cooling power, is simply $P_j E_j(W_j)$.

3.3 Utility Loss

Request routing. The concept of utility loss captures the lost revenue due to the user-perceived latency for request routing decisions. Latency is arguably the most important performance metric for most interactive services. A small increase in the user-perceived latency can cause substantial revenue loss for the provider [25]. We focus on the end-to-end propagation latency, which largely accounts for the user-perceived latency compared to other factors such as request processing times at datacenters [31]. The provider obtains the propagation latency L_{ij} between user i and datacenter j through active measurements [30] or other means.

We use α_{ij} to denote the volume of requests routed to datacenter j from user $i \in \mathcal{I}$, and D_i to denote the demand of each user that can be predicted using machine learning [28, 32]. Here, a user is an aggregated group of customers from a common geographical region, which may be identified by a unique IP prefix. The lost revenue from user i then depends on the *average* propagation latency $\sum_j \alpha_{ij} L_{ij} / D_i$ through a generic delay utility loss function U_i. U_i can take various forms depending on the interactive service. Our algorithm and proof work for general utility loss functions as long as U_i is increasing, differentiable, and convex.

As a case study, here we use a quadratic function to model user's increased tendency to leave the service with increased latency.

$$U_i(\alpha_i) = qD_i \left(\sum_{j \in \mathcal{J}} \alpha_{ij} L_{ij} / D_i \right)^2, \tag{2}$$

where q is the delay price that translates latency to monetary terms, and $\alpha_i = (\alpha_{i1}, \ldots, \alpha_{i|\mathcal{J}|})^T$. Utility loss is clearly zero when latency is zero between user and datacenter.

Capacity allocation. We denote the utility loss of allocating β_j servers for batch workloads as a differentiable, decreasing, and convex function $V_j(\beta_j)$, since allocating more resources increases the performance of batch jobs. Unlike interactive services, batch jobs are delay tolerant and resource elastic. Utility functions such as the log function are often used to capture such elasticity. However, utility functions model the benefit of resource allocation. To model the utility loss of resource allocation, since the loss is zero when the capacity is fully allocated to batch jobs, an intuitive definition can be of the following form:

$$V_j(\beta_j) = r(\log C_j - \log \beta_j), \tag{3}$$

where r is the utility price that converts the loss to monetary terms. (3) captures the intuition that increasing resources results in a decreasing marginal reduction of utility loss.

3.4 Problem Formulation

We now formulate the temperature aware workload management problem. For a given request routing decision α, the total cost associated with interactive workloads can be written as

$$\sum_{j \in \mathcal{J}} E_j \left(\sum_{i \in \mathcal{I}} \alpha_{ij} \right) P_j + \sum_{i \in \mathcal{I}} U_i(\alpha_i). \tag{4}$$

For a given capacity allocation decision β, the total cost associated with batch workloads is:

$$\sum_{j \in \mathcal{J}} E_j(\beta_j) P_j + \sum_{j \in \mathcal{J}} V_j(\beta_j). \tag{5}$$

Putting everything together, the optimization can be formulated as:

$$\begin{aligned} \text{minimize} \quad & (4) + (5) & (6) \\ \text{subject to:} \quad & \forall i : \sum_{j \in \mathcal{J}} \alpha_{ij} = D_i, & (7) \\ & \forall j : \sum_{i \in \mathcal{I}} \alpha_{ij} \le C_j - \beta_j, & (8) \\ & \alpha, \beta \succeq 0, & (9) \\ \text{variables:} \quad & \alpha \in \mathbb{R}^{|\mathcal{I}| \times |\mathcal{J}|}, \beta \in \mathbb{R}^{|\mathcal{J}|}. \end{aligned}$$

(6) is the objective function that jointly considers the cost of request routing and capacity allocation. (7) is the workload conservation constraint to ensure the user demand is satisfied. (8) is the datacenter capacity constraint, and (9) is the nonnegativity constraint.

3.5 Transforming to the ADMM Form

Problem (6) is a large-scale convex optimization problem. The number of users, i.e., unique IP prefixes, is typically $O(10^5)$–$O(10^6)$ for production systems. Hence, our problem can have tens of millions of variables, and millions of constraints. In such a setting, a distributed algorithm is preferable to fully utilize the computing resources of datacenters. Traditionally, dual decomposition with subgradient methods [9] are often used to develop distributed optimization algorithms. However, they suffer from the curse of step sizes. For the final output to be close to the optimum, we need to strategically pick the step size at each iteration, leading to well-known problems of slow convergence and performance oscillation with large-scale problems.

Alternating direction method of multipliers is a simple yet powerful algorithm that is able to overcome the drawbacks of dual decomposition methods, and is well suited to large-scale distributed convex optimization. Though developed in the 1970s [8], ADMM has recently received renewed interest, and found practical use in many large-scale distributed convex optimization problems in statistics, machine learning, etc. [10]. Before illustrating our new convergence proof and distributed algorithm that extend the classical framework, we first introduce the basics of ADMM, followed by a transformation of (6) to the ADMM form.

ADMM solves problems in the form

$$\begin{aligned} \min \quad & f_1(x_1) + f_2(x_2) & (10) \\ \text{s.t.} \quad & A_1 x_1 + A_2 x_2 = b, \\ & x_1 \in C_1, x_2 \in C_2, \end{aligned}$$

with variables $x_\ell \in \mathbb{R}^{n_\ell}$, where $A_\ell \in \mathbb{R}^{p \times n_\ell}$, $b \in \mathbb{R}^p$, f_ℓ's are convex functions, and C_ℓ's are non-empty polyhedral sets. Thus, the objective function is *separable* over *two* sets of variables, which are coupled through an equality constraint.

We can form the augmented Lagrangian [22] by introducing an extra L-2 norm term $\|A_1 x_1 + A_2 x_2 - b\|_2^2$ to the objective:

$$\begin{aligned} L_\rho(x_1, x_2; y) = f_1(x_1) + f_2(x_2) + y^T(A_1 x_1 + A_2 x_2 - b) \\ + (\rho/2)\|A_1 x_1 + A_2 x_2 - b\|_2^2. \end{aligned}$$

Here, $\rho > 0$ is the penalty parameter (L_0 is the standard Lagrangian for the problem). The benefits of introducing the penalty term are improved numerical stability and faster convergence in practice [10].

Our formulation (6) has a separable objective function due to the joint nature of the workload management problem. However, the request routing decision α and capacity allocation decision β are coupled by an inequality constraint rather than an equality constraint as in ADMM problems. Thus we introduce a slack variable $\gamma \in \mathbb{R}^{|\mathcal{J}|}$, and transform (6) to the following

$$\begin{aligned} \text{minimize} \quad & (4) + (5) + I_{\mathbb{R}_+^{|\mathcal{J}|}}(\gamma) && (11) \\ \text{subject to:} \quad & (7),\ (9), \\ & \forall j : \sum_i \alpha_{ij} + \beta_j + \gamma_j = C_j, && (12) \\ \text{variables:} \quad & \alpha \in \mathbb{R}^{|\mathcal{I}|\times|\mathcal{J}|}, \beta \in \mathbb{R}^{|\mathcal{J}|}, \gamma \in \mathbb{R}^{|\mathcal{J}|}. \end{aligned}$$

Here, $I_{\mathbb{R}_+^{|\mathcal{J}|}}(\gamma)$ is an indicator function defined as

$$I_{\mathbb{R}_+^{|\mathcal{J}|}}(\gamma) = \begin{cases} 0, & \gamma \succeq 0, \\ +\infty, & \text{otherwise.} \end{cases} \qquad (13)$$

The new formulation (11) is equivalent to (6), since for any feasible α and β, $\gamma \succeq 0$ holds, and the indicator function in the objective values to zero. Clearly, it is in the ADMM form, with a key difference that it has three sets of variables in the objective function and equality constraint (12). The convergence of the generalized m-block ADMM, where $m \geq 3$, has long remained an open question. Though it seems natural to directly extend the classical 2-block algorithm to the m-block case, such an algorithm may not converge unless some additional back-substitution step is taken [21]. Recently, some progresses have been made by [20, 23] that prove the convergence of m-block ADMM for strongly convex objective functions and the linear convergence of m-block ADMM under a full-column-rank relation matrix. However, the relation matrix in our setup is not full column rank. Thus, we need a new proof for the linear convergence under a general relation matrix, together with a distributed algorithm inspired by the proof.

4. THEORY

This section first introduces a generalized m-block ADMM algorithm inspired by [20, 23]. Then a new convergence proof is presented, which replaces the full column rank assumption with some mild assumptions on the objective function, and further simplifies the proof in [23]. The notations and discussions in this section are made intentionally independent of the other parts of the paper in order to present the proof in a mathematically general way.

4.1 Algorithm

We consider a convex optimization problem in the form

$$\begin{aligned} \min \quad & \sum_{i=1}^m f_i(x_i) && (14) \\ \text{s.t.} \quad & \sum_{i=1}^m A_i x_i = b \end{aligned}$$

with variables $x_i \in \mathbb{R}^{n_i}$ $(i = 1, \ldots, m)$, where $f_i : \mathbb{R}^{n_i} \to \mathbb{R}$ $(i = 1, \ldots, m)$ are closed proper convex functions; $A_i \in \mathbb{R}^{l\times n_i}$ $(i = 1, \ldots, m)$ are given matrices; and $b \in \mathbb{R}^l$ is a given vector.

We form the augmented Lagrangian

$$L_\rho(x_1, \ldots, x_m; y) = \sum_{i=1}^m f_i(x_i) + y^T(\sum_{i=1}^m A_i x_i - b) + (\rho/2)\|\sum_{i=1}^m A_i x_i - b\|_2^2. \quad (15)$$

As in [23], a generalized ADMM algorithm has the following:

$$x_i^{k+1} = \operatorname*{argmin}_{x_i} L_\rho(x_1^{k+1}, \ldots, x_{i-1}^{k+1}, x_i, x_{i+1}^k, \ldots, x_m^k; y^k), \quad i = 1, \ldots, m,$$

$$y^{k+1} = y^k + \varrho(\sum_{i=1}^m A_i x_i^{k+1} - b),$$

where $\varrho > 0$ is the step size for the dual update. Note that when $m = 2$ and the step size ϱ equals to the penalty parameter ρ, the above algorithm is reduced to the standard ADMM algorithm presented in [8].

4.2 Assumptions

We present two assumptions on the objective functions, based on which we are able to show the convergence of the generalized m-block ADMM algorithm.

ASSUMPTION 1. *The objective functions f_i $(i = 1, \ldots, m)$ are strongly convex.*

Note that strong convexity is quite reasonable in engineering practice. This is because a convex function $f(x)$ can be always well-approximated by a strongly convex function $\bar{f}(x)$. For instance, if we choose $\bar{f}(x) = f(x) + \epsilon\|x\|_2^2$ for some sufficiently small $\epsilon > 0$, then $\bar{f}(x)$ is strongly convex.

ASSUMPTION 2. *The gradients ∇f_i $(i = 1, \ldots, m)$ are Lipschitz continuous.*

Assumption 2 says that, for each i, there exists some constant $\kappa_i > 0$ such that for all $x_1, x_2 \in \mathbb{R}^{n_i}$,

$$\|\nabla f_i(x_1) - \nabla f_i(x_2)\|_2 \leq \kappa_i \|x_1 - x_2\|_2,$$

which is again reasonable in practice, since κ_i can be made sufficiently large.

4.3 Convergence

In this section, we outline the proof for the convergence of the generalized ADMM algorithm. The detailed proof can be found in Sec. 4.3 of our technical report [39].

For convenience, we write

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}, \ f(x) = \sum_{i=1}^m f_i(x_i), \text{ and } A = [A_1 \ \ldots \ A_m].$$

Then the problem (14) can be rewritten as

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & Ax = b \end{aligned}$$

with the optimal value $p^* = \inf\{f(x) \mid Ax = b\}$. Similarly, the augmented Lagrangian can be rewritten as

$$L_\rho(x; y) = f(x) + y^T(Ax - b) + (\rho/2)\|Ax - b\|_2^2,$$

with the associated dual function defined by

$$d(y) = \inf_x L_\rho(x; y)$$

and the optimal value $d^* = \sup\{d(y)\}$.

Now define the primal and dual optimality gaps as

$$\Delta_p^k = L_\rho(x^{k+1}; y^k) - d(y^k),$$
$$\Delta_d^k = d^* - d(y^k),$$

respectively. Clearly, we have $\Delta_p^k \geq 0$ and $\Delta_d^k \geq 0$. Define

$$V^k = \Delta_p^k + \Delta_d^k.$$

We will see that V^k is a *Lyapunov function* for the algorithm, *i.e.*, a nonnegative quantity that decreases in each iteration.

Our proof relies on three technical lemmas.

LEMMA 1. *There exists a constant $\vartheta > 0$ such that*

$$V^k \leq V^{k-1} - \varrho\|A\bar{x}^{k+1} - b\|_2^2 - \vartheta\|x^{k+1} - x^k\|_2^2, \quad (16)$$

in each iteration, where $\bar{x}^{k+1} = \mathrm{argmin}_x L_\rho(x; y^k)$.

PROOF. See Appendix C in the technical report [39]. □

LEMMA 2. *For any given $\delta > 0$, there exists a constant $\tau > 0$ (depending on δ) such that for any (x, y) satisfying $\|x\| + \|y\| \leq 2\delta$, the following inequality holds*

$$\|x - \bar{x}(y)\| \leq \tau\|\nabla_x L_\rho(x; y)\|, \quad (17)$$

where $\bar{x}(y) = \arg\min_x L_\rho(x; y)$.

PROOF. See Appendix B in the technical report [39]. □

LEMMA 3. *There exists a constant $\eta > 0$ such that*

$$\|\nabla_x L_\rho(x^k; y^k)\|_2 \leq \eta\|x^k - x^{k+1}\|_2. \quad (18)$$

PROOF. See Appendix A in the technical report [39]. □

By Lemma 1, we have

$$\sum_{k=0}^{\infty} \left(\varrho\|A\bar{x}^{k+1} - b\|_2^2 + \vartheta\|x^{k+1} - x^k\|_2^2\right) \leq V^0.$$

Hence, $\|A\bar{x}^{k+1} - b\|_2^2 \to 0$ and $\|x^{k+1} - x^k\|_2^2 \to 0$, as $k \to \infty$. Suppose that the level set of $\Delta_p + \Delta_d$ is bounded. Then by the Bolzano-Weierstrass theorem, the sequence $\{x^k, y^k\}$ has a convergent subsequence, *i.e.*,

$$\lim_{k \in \mathcal{R}, k \to \infty} (x^k, y^k) = (\tilde{x}, \tilde{y}),$$

for some subsequence $\mathcal{R}$, where $(\tilde{x}, \tilde{y})$ denotes the limit point. By using Lemma 2 and Lemma 3, we can show that the limit point $(\tilde{x}, \tilde{y})$ is an optimal primal-dual solution. Hence,

$$\lim_{k \in \mathcal{R}, k \to \infty} V^k = \lim_{k \in \mathcal{R}, k \to \infty} \Delta_p^k + \Delta_d^k = 0.$$

Since V^k decreases in each iteration, the convergence of a subsequence of V^k implies the convergence of V^k, and we have

$$\lim_{k \to \infty} \Delta_p^k + \Delta_d^k = 0.$$

This further implies that both Δ_p^k and Δ_d^k converge to 0.

To sum up, we have the following convergence theorem for our generalized ADMM algorithm.

THEOREM 1. *Suppose that Assumptions 1 and 2 hold and that the level set of $\Delta_p + \Delta_d$ is bounded. Then both the primal gap Δ_p^k and the dual gap Δ_d^k converge to 0.*

Due to space limit, the rate of convergence is omitted and can be found in Sec. 4.3 of [39].

5. A DISTRIBUTED ALGORITHM

We now develop a distributed solution algorithm based on the generalized ADMM algorithm in Sec. 4.1. Directly applying the algorithm to our problem (11) will lead to a centralized algorithm. The reason is that when the augmented Lagrangian is minimized over α, the penalty term $\sum_j \Big(\sum_i \alpha_{ij} + \beta_j + \gamma_j - C_j\Big)^2$ couples α_{ij}'s across i, and the utility loss $\sum_i U_i(\alpha_i)$ couples α_{ij}'s across j. The joint optimization of utility loss and the quadratic penalty is particularly difficult to solve, especially when the number of users is large, since $U_i(\alpha_i)$ can take any general form. If they can be separated, then we will have a distributed algorithm where each $U_i(\alpha_i)$ is optimized in parallel, and the quadratic penalty term is optimized efficiently with existing methods.

Towards this end, we introduce a new set of auxiliary variables $a_{ij} = \alpha_{ij}$, and re-formulate the problem (11):

$$\begin{aligned} \text{minimize} \quad & \sum_j E_j(\sum_i a_{ij})P_j + \sum_i U_i(\alpha_i) + (5) + I_{\mathbb{R}_+^{|\mathcal{J}|}}(\gamma) \\ \text{subject to:} \quad & (7),\ (9), \\ & \forall j : \sum_i a_{ij} + \beta_j + \gamma_j = C_j, \\ & \forall i, j : a_{ij} = \alpha_{ij}, \\ \text{variables:} \quad & a, \alpha \in \mathbb{R}^{|\mathcal{I}| \times |\mathcal{J}|}, \beta, \gamma \in \mathbb{R}^{|\mathcal{J}|}. \end{aligned} \quad (19)$$

This is a 4-block ADMM problem, where a_{ij} replaces α_{ij} in the objective function and constraint (12) when the coupling happens across users i. This is the key step that enables the decomposition of the α-minimization problem. The augmented Lagrangian can then be readily obtained from (15). By omitting the irrelevant terms, we can see that at each iteration $k+1$, the α-minimization problem is

$$\begin{aligned} \min \quad & \sum_i U_i(\alpha_i) - \sum_j \sum_i \Big(\varphi_{ij}\alpha_{ij} - \frac{\rho}{2}(\alpha_{ij}^2 - 2\alpha_{ij}a_{ij}^k)\Big) \\ \text{s.t.} \quad & \forall i : \sum_j \alpha_{ij} = D_i, \alpha_i \succeq 0, \end{aligned} \quad (20)$$

where φ_{ij} is the dual variable for the equality constraint $a_{ij} = \alpha_{ij}$. This is clearly decomposable over i into $|\mathcal{I}|$ per-user sub-problems since the objective function and constraint are separable over i. The per-user sub-problem is of a much smaller scale with only $|\mathcal{J}|$ variables and $|\mathcal{J}|+1$ constraints, and is easy to solve even though it is a non-linear problem for a general U_i.

Some may now wonder if the auxiliary variable a is hard to solve for. As it turns out, the a-minimization problem is decomposable over j into $|\mathcal{J}|$ per-datacenter sub-problems. Moreover, each per-datacenter sub-problem is a quadratic program. Though it is large-scale, it can be transformed into a second-order cone program and solved efficiently. More details can be found in Sec. 5 in the technical report [39].

β- and γ-minimization steps are clearly decomposable over j. The entire procedure is summarized below.

Distributed 4**-block ADMM**. Initialize $a, \alpha, \beta, \gamma, \lambda, \varphi$ to 0. For $k = 0, 1, \dots,$ repeat

1. α**-minimization:** Each user solves the following sub-problem for α_i^{k+1}:

$$\begin{aligned} \min \quad & U_i(\alpha_i) - \sum_j \Big(\varphi_{ij}\alpha_{ij} - \frac{\rho}{2}(\alpha_{ij}^2 - 2\alpha_{ij}a_{ij}^k)\Big) \\ \text{s.t.} \quad & \sum_j \alpha_{ij} = D_i, \alpha_i \succeq 0. \end{aligned} \tag{21}$$

2. a**-minimization:** Each datacenter solves the following sub-problem for $a_j^{k+1} = (a_{1j}^{k+1}, \dots, a_{|\mathcal{I}|j}^{k+1})^T$:

$$\begin{aligned} \min E_j\Big(\sum_i a_{ij}\Big)P_j &+ \sum_i a_{ij}(\lambda_j^k + \varphi_{ij}^k) + \frac{\rho}{2}(\sum_i a_{ij})^2 \\ &+ \rho\Big(\sum_i a_{ij}(\beta_j^k + \gamma_j^k - C_j + 0.5a_{ij} - \alpha_{ij}^{k+1})\Big) \\ \text{s.t. } a_j \succeq 0. & \end{aligned} \tag{22}$$

3. β**-minimization:** Each datacenter solves the following sub-problem for β_j^{k+1}:

$$\begin{aligned} \min \quad & E_j(\beta_j)P_j + V_j(\beta_j) + \lambda_j^k\beta_j \\ & + \frac{\rho}{2}\Big(\sum_i a_{ij}^{k+1} + \beta_j + \gamma_j^k - C_j\Big)^2 \\ \text{s.t.} \quad & \beta_j \geq 0. \end{aligned}$$

4. γ**-minimization:** Each datacenter solves:

$$\gamma_j^{k+1} = \max\left\{0, C_j - \frac{\lambda_j}{\rho} - \sum_i a_{ij}^{k+1} - \beta_j^{k+1}\right\}, \forall j.$$

5. **Dual update:** Each datacenter updates λ_j for the capacity constraint (8):

$$\lambda_j^{k+1} = \lambda_j^k + \varrho\Big(\sum_i a_{ij}^{k+1} + \beta_j^{k+1} + \gamma_j^{k+1} - C_j\Big).$$

Each user updates φ_{ij} for the equality constraint $a_{ij} = \alpha_{ij}$:

$$\varphi_{ij}^{k+1} = \varphi_{ij}^k + \varrho(a_{ij}^{k+1} - \alpha_{ij}^{k+1}), \forall j.$$

The distributed nature of our algorithm allows for an efficient parallel implementation in datacenters with a large number of servers. The per-user sub-problem (21) can be solved in parallel on each server. Since (21) is a small-scale convex optimization as discussed above, the complexity is low. A multi-threaded implementation can further speed up the algorithm with multi-core hardware. The penalty parameter ρ and utility loss function U_i can be configured at each server before the algorithm runs. Step 2 and 3 involve solving $|\mathcal{J}|$ per-datacenter sub-problems respectively, which can also be implemented in parallel with only $|\mathcal{J}|$ servers.

6. EVALUATION

We perform trace-driven simulations to realistically assess the potential of temperature aware workload management.

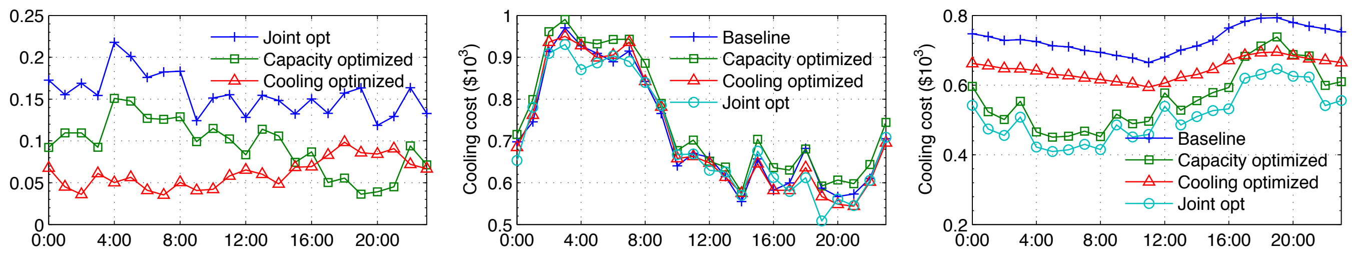
6.1 Setup

We rely on the Wikipedia request traces [38] to represent the interactive workloads of a cloud service. The dataset we use contains, among other things, 10% of all user requests issued to Wikipedia from the 24-hour period between January 1, 2008 UTC to January 2, 2008 UTC. The workloads are normalized to a number of servers, assuming that each request requires 10% of a server's CPU. The traces reflect the diurnal pattern of real-world interactive workloads. The prediction of workloads can be done accurately as demonstrated by previous work [28, 32], and we do not consider the effect of prediction error here. The optimization is solved hourly.

We consider Google's infrastructure [4] to represent a geo-distributed cloud as discussed in Sec. 2.3. Each datacenter's capacity C_j is uniformly distributed between $[1, 2] \times 10^5$ servers. The empirical CRAC efficiency model developed in Sec. 3.2 is used to derive the total energy consumption of all 13 locations under different temperatures. We use the 2011 annual average day-ahead on peak prices [16] at the local markets as the power prices P_j for the 6 U.S. locations[3]. For non-U.S. locations, the power price is calculated based on the retail industrial power price available on the local utility company websites with a 50% wholesale discount, which is usually the case in reality [37]. The power prices at each location are shown in Table 2 in the technical report [39]. The servers have peak power $P_{\text{peak}} = 200$ W, and consume 50% power at idle. These numbers represent state-of-the-art datacenter hardware [15, 34].

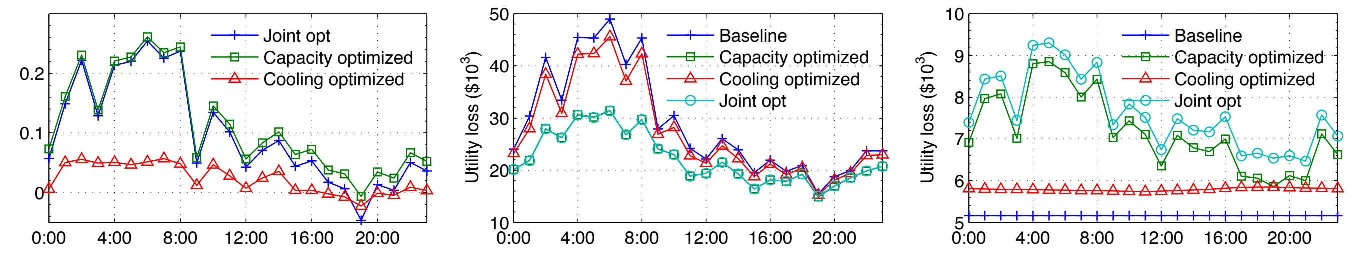
To calculate the utility loss of interactive workloads, we obtain the latency matrix L from iPlane [30], a system that collects wide-area network statistics from Planetlab vantage points. Since the Wikipedia traces do not contain client side information, we emulate the geographical diversity of user requests by splitting the total interactive workloads among users following a normal distribution. We set the number of

[3]The U.S. electricity market is consisted of multiple regional markets. Each regional market has several hubs with their own pricing. We thus use the price of the specific hub that each U.S. datacenter locates in.



(a) Overall improvement. (b) Interactive workloads. (c) Batch workloads.

Figure 4: Cooling energy cost savings. Time is in UTC.



(a) Overall improvement. (b) Interactive workloads. (c) Batch workloads.

Figure 5: Utility loss reductions. Time is in UTC.

users $|\mathcal{I}| = 10^5$, and choose 10^5 IP prefixes from a Route-Views [5] dump. Note that in our context, each user, i.e. IP prefix, represents many customers accessing the service. We then extract the corresponding round trip times from iPlane logs, which contain traceroutes made to IP addresses from Planetlab nodes. We only use latency measurements from Planetlab nodes that are close to our datacenter locations to resemble the user-datacenter latency. We use utility loss functions defined in (2) and (3). The delay price $q = 4 \times 10^{-6}$, and the utility loss price for batch jobs $r = 500$.

We investigate the performance of temperature aware workload management. We benchmark our ADMM algorithm, referred to as *Joint opt*, against three baseline strategies, which use different amounts of information in managing workloads.

The first benchmark, called *Baseline*, is a temperature agnostic strategy that separately considers capacity allocation and request routing of the workload management problem. It first allocates capacity to batch jobs by minimizing the back-end total cost with (5) as the objective. The remaining capacity is used to solve the request routing optimization with (4) as the objective. Only the electricity price diversity is used, and cooling energy is calculated with a constant pPUE of 1.2 that corresponds to an ambient temperature of 20°C for the two cost minimization problems. Though naive, such an approach is widely used in current Internet-scale cloud services. It also allows an implicit comparison with prior work [17, 27, 29, 34, 35].

The second benchmark, called *Capacity Optimized*, improves upon *Baseline* by jointly solving capacity allocation and request routing, but still ignores the cooling energy efficiency diversity. This demonstrates the impact of capacity allocation in datacenter workload management.

The third benchmark, called *Cooling Optimized*, improves upon *Baseline* by exploiting the temperature and cooling efficiency diversity in minimizing cost, but does not adopt joint management of the interactive and batch workloads. This demonstrates the impact of being temperature aware.

We run the four benchmarks above with our 24-hour traces at each day of January 2011, using the empirical hourly temperature data we collected in Sec. 2.3. The distributed ADMM algorithm is used to solve them until convergence is achieved. The figures show the average results over 31 runs.

6.2 Cooling energy savings

The central thesis of this paper is to save datacenter cost through temperature aware workload management that exploits the cooling efficiency diversity with capacity allocation. We examine the effectiveness of our approach by comparing the cooling energy consumption first. Figure 4 shows the results.

In particular, Figure 4a shows that overall, *Joint opt* saves 15%–20% cooling energy compared to *Baseline*. A breakdown of the saving shown in the same figure reveals that dynamic capacity allocation provides 10%–15% saving, and cooling efficiency diversity provides 5%–10% saving, respectively. Note that the cost saving is achieved with cutting-edge CRACs whose efficiency is already substantially improved with outside air cooling capability. The results confirm that our temperature aware workload management is able to further optimize the cooling efficiency and cost of geo-distributed datacenters.

Figure 4b and 4c show a detailed breakdown of cooling energy cost. Cooling cost attributed to interactive workloads, as in Figure 4b, exhibits a diurnal pattern and peaks between 2:00 and 8:00 UTC (21:00 to 3:00 EST, 18:00 to 0:00 PST), implying that most of the Wikipedia traffic originates from the U.S. The four strategies perform fairly closely, while *Baseline* and *Capacity optimized* consistently incur more cooling energy cost due to their cooling agnostic nature that underestimates the overall energy cost.

Cooling cost attributed to batch workloads is shown in Figure 4c. *Baseline* incurs the highest cost since it underestimates the energy cost, and runs more batch workloads than necessary. *Cooling optimized* improves *Baseline* by taking into account cooling efficiency diversity and reducing batch workloads as a result. Both strategies fail to exploit the trade-off with interactive workloads. Thus their cooling cost closely follows the daily temperature trend in that it gradually decreases from 0:00 to 12:00 UTC (19:00 to 7:00 EST) and then slowly increases from 12:00 to 20:00 UTC (7:00 to 15:00 EST). *Capacity optimized* adjusts capacity allocation with request routing, and further reduces batch workloads in order to allocate more resources for interactive workloads. *Joint opt* combines temperature aware cooling optimization with holistic workload management, and has the lowest cooling cost with least batch workloads. Though this increases the back-end utility loss, the overall effect is a net reduction of total cost since interactive workloads enjoy lower latency as will be observed soon.

6.3 Utility loss reductions

The other component of datacenter cost is utility loss. From Figure 5a, the relative reduction follows the interactive workloads and also has a visible diurnal pattern. *Joint opt* and *Capacity optimized* provide the most significant utility loss reductions from 5% to 25%, while *Cooling optimized* provides a modest 5% reduction compared to *Baseline*. To study the reasons for the varying degrees of reductions, Figure 5b and 5c show the respective utility loss of interactive and batch workloads. We observe that interactive workloads incur most of the utility loss, reflecting its importance compared to batch workloads. *Baseline* and *Cooling optimized* have much larger utility loss from interactive workloads as shown in Figure 5b, because of the separate management of two workloads. The average latency performances under these two strategies are also worse as can be seen in Figure 7 of our technical report [39].

On the other hand, *Capacity optimized* and *Joint opt* outperform the two by allocating more capacity to interactive workloads at cost-efficient locations while reducing batch workloads (recall Figure 4c). This is especially effective during peak hours as shown in Figure 5b. *Capacity optimized* and *Joint opt* do have larger utility loss from batch workloads as seen in Figure 5c. However since interactive workloads attribute to the majority of the provider's utility and revenue, the overall effect of joint workload management is positive.

6.4 Sensitivity to seasonal changes

One natural question is, since the results above are obtained in winter times (January), would the benefits be less significant during summer times when cooling is more expensive? In other words, are the benefits sensitive to the seasonal changes? We thus run our *Joint opt* with *Baseline* at each day of May, which represents typical Spring/Fall weather, and August, which represents typical Summer weather, respectively. Figure 6 shows the average overall cost savings achieved in different seasons. We observe that the cost savings, ranging from 5% to 20%, are consistent and insensitive to seasonal changes. The reason is that our approach depends on: 1) the geographical diversity of temperature and cooling efficiency; 2) the mixed nature of datacenter workloads, both of which exist at all times of the year no matter which cooling method is used. Temperature aware workload management is thus able to offer consistent cost benefits.

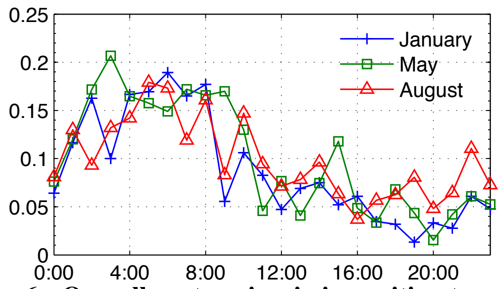


Figure 6: Overall cost saving is insensitive to seasonal changes of the climate.

We also compare the convergence speed of our the distributed ADMM algorithm with the conventional subgradient method. We have found that our algorithm converges within around 60 iterations, while the subgradient method does not converge even after 200 iterations. Our distributed ADMM algorithm is thus better suited to large-scale convex optimization problems. More details can be found in Sec. 6.3 in the technical report [39].

7. CONCLUSION

We propose temperature aware workload management, which explores two key aspects of geo-distributed datacenters that have not been well understood in the past. First, as we show empirically, energy efficiency of cooling systems, especially outside air cooling, varies widely with outside temperature. The geographical diversity of temperature is utilized to reduce cooling energy consumption. Second, the elastic nature of batch workloads is further capitalized by dynamically adjusting capacity allocation along with the widely studied request routing for interactive workloads. We formulate the joint optimization under a general framework with an empirical cooling efficiency model. To solve large-scale problems for production systems, we rely on the ADMM algorithm. We provide a new convergence proof for a generalized m-block ADMM algorithm. We further develop a novel distributed ADMM algorithm for our problem. Extensive simulations highlight that temperature aware workload management saves 15%–20% cooling energy and 5%–20% overall energy cost and the distributed ADMM algorithm is practical to solve large-scale workload management problems with only tens of iterations.

8. REFERENCES

[1] http://tinyurl.com/89ros64.

[2] http://tinyurl.com/8ulxfzp.

[3] http://tinyurl.com/bpqv6tl.

[4] https://www.google.com/about/datacenters/inside/locations/.

[5] http://www.routeviews.org.

[6] National climate data center (NCDC). http://www.ncdc.noaa.gov.

[7] BASH, C., AND FORMAN, G. Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *Proc. USENIX ATC* (2007).

[8] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[9] BOYD, S., AND MUTAPCIC, A. Subgradient methods. Lecture notes of EE364b, Stanford University, Winter Quarter 2006-2007. http://www.stanford.edu/class/ee364b/notes/subgrad_method_notes.pdf.

[10] BOYD, S., PARIKH, N., CHU, E., PELEATO, B., AND ECKSTEIN, J. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning 3*, 1 (2010), 1–122.

[11] CHEN, Y., GMACH, D., HYSER, C., WANG, Z., BASH, C., HOOVER, C., AND SINGHAL, S. Integrated management of application performance, power and cooling in datacenters. In *Proc. NOMS* (2010).

[12] DENG, N., STEWART, C., GMACH, D., ARLITT, M., AND KELLEY, J. Adaptive green hosting. In *Proc. ACM ICAC* (2012).

[13] EL-SAYED, N., STEFANOVICI, I., AMVROSIADIS, G., AND HWANG, A. A. Temperature management in data centers: Why some (might) like it hot. In *Proc. ACM Sigmetrics* (2012).

[14] EMERSON NETWORK POWER. Liebert® DSE™ precision cooling system sales brochure. http://tinyurl.com/c7e8qxz, 2012.

[15] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power provisioning for a warehouse-sized computer. In *Proc. ACM/IEEE Intl. Symp. Computer Architecture (ISCA)* (2007).

[16] FEDERAL ENERGY REGULATORY COMMISSION. U.S. electric power markets. http://www.ferc.gov/market-oversight/mkt-electric/overview.asp, 2011.

[17] GAO, P. X., CURTIS, A. R., WONG, B., AND KESHAV, S. It's not easy being green. In *Proc. ACM SIGCOMM* (2012).

[18] GOIRI, I. N., BEAUCHEA, R., LE, K., NGUYEN, T. D., HAQUE, M. E., GUITART, J., TORRES, J., AND BIANCHINI, R. Greenslot: Scheduling energy consumption in green datacenters. In *Proc. SC* (2011).

[19] GOIRI, I. N., LE, K., NGUYEN, T. D., GUITART, J., TORRES, J., AND BIANCHINI, R. GreenHadoop: Leveraging green energy in data-processing frameworks. In *Proc. ACM EuroSys* (2012).

[20] HAN, D., AND YUAN, X. A note on the alternating direction method of multipliers. *J. Optim. Theory Appl. 155* (2012), 227–238.

[21] HE, B. S., TAO, M., AND YUAN, X. M. Alternating direction method with Gaussian back substitution for separable convex programming. *SIAM J. Optim. 22* (2012), 313–340.

[22] HESTENES, M. R. Multiplier and gradient methods. *Journal of Optimization Theory and Applications 4*, 5 (1969), 303–320.

[23] HONG, M., AND LUO, Z.-Q. On the linear convergence of the alternating direction method of multipliers, August 2012.

[24] INTEL INC. Reducing data center cost with an air economizer, August 2008.

[25] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proc. ACM SIGKDD* (2007).

[26] LE, K., BIANCHINI, R., NGUYEN, T. D., BILGIR, O., AND MARTONOSI, M. Capping the brown energy consumption of Internet services at low cost. In *Proc. IGCC* (2010).

[27] LIN, M., WIERMAN, A., ANDREW, L. L. H., AND THERESKA, E. Dynamic right-sizing for power-proportional data centers. In *Proc. IEEE INFOCOM* (2011).

[28] LIU, Z., CHEN, Y., BASH, C., WIERMAN, A., GMACH, D., WANG, Z., MARWAH, M., AND HYSER, C. Renewable and cooling aware workload management for sustainable data centers. In *Proc. ACM Sigmetrics* (2012).

[29] LIU, Z., LIN, M., WIERMAN, A., LOW, S. H., AND ANDREW, L. L. Greening geographical load balancing. In *Proc. ACM Sigmetrics* (2011).

[30] MADHYASTHA, H. V., ISDAL, T., PIATEK, M., DIXON, C., ANDERSON, T., KRISHNAMURTHY, A., AND VENKATARAMANI, A. iPlane: An information plane for distributed services. In *Proc. USENIX OSDI* (2006).

[31] NARAYANA, S., JIANG, J. W., REXFORD, J., AND CHIANG, M. To coordinate or not to coordinate? Wide-Area traffic management for data centers. Tech. rep., Princeton University, 2012.

[32] NIU, D., XU, H., LI, B., AND ZHAO, S. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications. In *Proc. IEEE INFOCOM* (2012).

[33] PELLEY, S., MEISNER, D., WENISCH, T. F., AND VANGILDER, J. W. Understanding and abstracting

total data center power. In *Proc. Workshop on Energy Efficient Design (WEED)* (2009).

[34] QURESHI, A., WEBER, R., BALAKRISHNAN, H., GUTTAG, J., AND MAGGS, B. Cutting the electricity bill for Internet-scale systems. In *Proc. ACM SIGCOMM* (2009).

[35] RAO, L., LIU, X., XIE, L., AND LIU, W. Minimizing electricity cost: Optimization of distributed Internet data centers in a multi-electricity-market environment. In *Proc. IEEE INFOCOM* (2010).

[36] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. ACM SoCC* (2012).

[37] TELEGEOGRAPHY RESEARCH. Global Internet geography executive summary. `http://bpastudio.csudh.edu/fac/lpress/471/hout/telegeographygig_execsumm.pdf`, 2008.

[38] URDANETA, G., PIERRE, G., AND VAN STEEN, M. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks 53*, 11 (July 2009), 1830–1845.

[39] XU, H., FENG, C., AND LI, B. Temperature aware workload management in geo-distributed datacenters. Tech. rep., University of Toronto, `http://iqua.ece.toronto.edu/~henryxu/share/geodc-preprint.pdf`, 2013.

[40] ZHOU, R., WANG, Z., MCREYNOLDS, A., BASH, C., CHRISTIAN, T., AND SHIH, R. Optimization and control of cooling microgrids for data centers. In *Proc. IEEE ITherm* (2012).

Power-Aware Throughput Control for Database Management Systems

Zichen Xu
The Ohio State University
xuz@ece.osu.edu

Xiaorui Wang
The Ohio State University
xwang@ece.osu.edu

Yi-Cheng Tu
University of South Florida
ytu@cse.usf.edu

Abstract

Performance has been traditionally regarded as the most important design goal for database management systems (DBMSs). However, in recent years, the increasing energy cost gradually rivals the benefit of chasing after performance. Therefore, there are strong financial incentives to minimize power consumption of a database system while maintaining its desired performance, so that the energy cost can be best amortized. Such a goal is challenging in practice because the power consumption of a database system varies significantly with the environment and workloads. Many modern hardware provide multiple modes with different power/performance tradeoffs. However, existing research has not used these power modes sufficiently to achieve the best tradeoff for database services due to the lack of the knowledge on database behavior under different power modes. In this paper, we present Power-Aware Throughput control (PAT), an online feedback control framework for energy conservation at the DBMS level. In contrast to heuristic-based tuning techniques commonly used in database systems, the design of PAT is based on rigorous control-theoretic analysis for guaranteed control accuracy and system stability. We implement PAT as an integrated component of the PostgreSQL system and evaluate it with workloads generated from various database benchmarks. The results show that PAT achieves up to 51.3% additional energy savings despite runtime workload dynamics and model errors, as compared to other competing methods.

1 Introduction

The rapid growth of energy-related research in databases is driven by the fact that *data centers are energy starving*. The increasing operating expenses of data centers (e.g., the electricity bill) quickly deplete the revenue earned from database services due to its accumulating demand of energy [18]. The power-performance tradeoff has now become a new key challenge in general purpose database system design [30].

Redesigning DBMS towards high energy efficiency has been discussed in the database community. Poess et al. [19] examine the power saving opportunities from different hardware systems. Lang et al. [11] report large energy savings by using the dynamic voltage and frequency scaling (DVFS) technique in CPUs. However, it is not a trivial task to harvest those opportunities in data processing while maintaining the desired performance . The DBMS performance could be very sensitive to the changes in hardware power modes. For example, tuning one step (25%) down in CPU frequency could result in about 30% performance degradation for CPU intensive queries; in addition, switching low-power modes in memory is a bad idea due to significant performance degradation for any DBMS queries, as shown in Fig.1, Section 2. Therefore, we cannot directly apply existing hardware power management techniques in DBMSs for the energy conservation.

It is also difficult to provide performance guarantees in a DBMS due to workload variations and environment dynamics. We need an adaptive architecture that could promptly monitor query statistics from DBMS and determine whether/to what extend adaption should be performed. Attempting to solve the problem, some studies employ simple hill-climbing strategies to make such important adaption decision [11, 10]. These *ad hoc* control solutions cannot provide desired control performance, such as zero steady-state error and short settling time bound [4]. Although there are many control work done at the OS level, such as [27, 26, 17], they are not feasible due to the lack of critical database information that is needed for making adaptation decisions.

To address the aforementioned problems, we first need to understand the nature of the DBMS's response to the changes of different hardware power modes ("knobs"). Specifically, we need a quantitative system model in the adaptive framework that describes how the DBMS performance changes in response to knobs tuning. Second, the adaptive framework needs to be implemented in light weight without affecting the normal DBMS operation. Finally, the control algorithm shall be robust such that it could tolerate errors from estimation in the DBMS optimizer and workload variations.

In this paper, we present Power-Aware Throughput control (PAT), an online feedback control framework for energy conservation at the DBMS level, to address the above challenges. Our solution takes advantage of well-established techniques from the field of control theory, to deal with systems that are subject to unpredictable dynamics [4]. In this solution, we formulate energy conservation with performance control at the DBMS level into a feedback control problem and tackle it with a proportional-integral (PI) controller based on the DBMS system model. Specifically, this paper makes the following contributions:

- We explore the relationship among query statistics, the DBMS throughput, hardware power states, and the active power consumption[1] via empirical studies. Our results show that 1) there exists great energy savings when tuning DVFS for processing I/O intensive queries; 2) The relationship between the DBMS throughput and the CPU frequency is an approximated linear model when DBMS workloads are steady; 3) the ratio of I/O intensive queries in the workload plays a major role in the workload statistics that affect the performance of the control.

- As one of the first attempts to introduce classic control theory into the energy management in DBMSs, we design PAT to control the DBMS throughput while minimizing the active power consumption.

- We design and implement a query classifier based on the fuzzy set theory. The classifier provides important information, such as the ratio of I/O queries, which plays a key role in achieving effective throughput control. The fuzzy-logic-based design also provides new insights to the classic problem of query clustering.

- We implement PAT within the real DBMS – PostgreSQL and evaluate it with various baselines. The results show that, PAT has significantly more energy saving (51.3%) with the least control errors comparing with other control baselines.

The rest of the paper is organized as follows: we first discuss our study on characterization of database system in Section 2. Section 3 introduces the overall control framework; Sections 4 and 5 present the design and analysis of the workload classifier and controller in PAT, respectively. Section 6 talks about our empirical evaluation of the proposed control strategy. Section 7 compares our work with related work; Section 8 concludes the paper.

[1] we use the active power of the whole system for the measurement throughout this paper. Any power data, if without specification, is the active power of the system.

2 System Characterization Study

In this section, we report our findings based on empirical studies of database behavior as a foundation of control framework design.[2]

In our study, we focus on the DBMS throughput (query per second, QPS) as the main performance metric. The throughput, as the reciprocal of the average response time, is an important performance metric. For example, transaction processing performance council (TPC) uses throughput to define and rank the performance of different DBMS products [22]. To keep the DBMS throughput within a desired level is essential to avoid situations, such as overloading. We take controlling the response time of individual queries as a future work for the design of PAT, which will not be discussed in this paper.

The impact of hardware power modes with different DBMS workloads: to further understand the impact of low-power modes in different hardware components on the power consumption and the performance of database services, we use five power states of the memory (described in [3]), four discrete DVFS levels of the CPU (described in [27]), and the CPU C-state (described in [15] and labeled as "DVFS0"). To avoid possible bias from measurement errors, we repeat experiments using CPU intensive and I/O intensive workloads in several trials and collect the average result, demonstrated in Fig.1.

Fig.1(a) and Fig.1(b) show the DBMS performance and the power measurement of different power states in memory under two types of DBMS workloads. As we can see, a state transition in memory, such as from the active state to the active-standby state, can contribute to at most a 10% saving in active power. However, the power saving comes with a severe performance penalty as a 95% performance degradation in CPU workloads and a 98% degradation in I/O workloads after the transition. The penalty comes from unacceptable low I/O bandwidths from memory low power modes, which make any processing queries enter infinite cycles of I/O wait. Thus, although [3] claims energy savings from tuning power states in the memory, it may not be a feasible solution for database services. As a result, we find that any hardware power management techniques which increase per-page I/O cost may have a severe consequence on the DBMS throughput, which eventually leads to unacceptable high energy cost.

[2] details of the experiment setup can be found in Section 6.1.

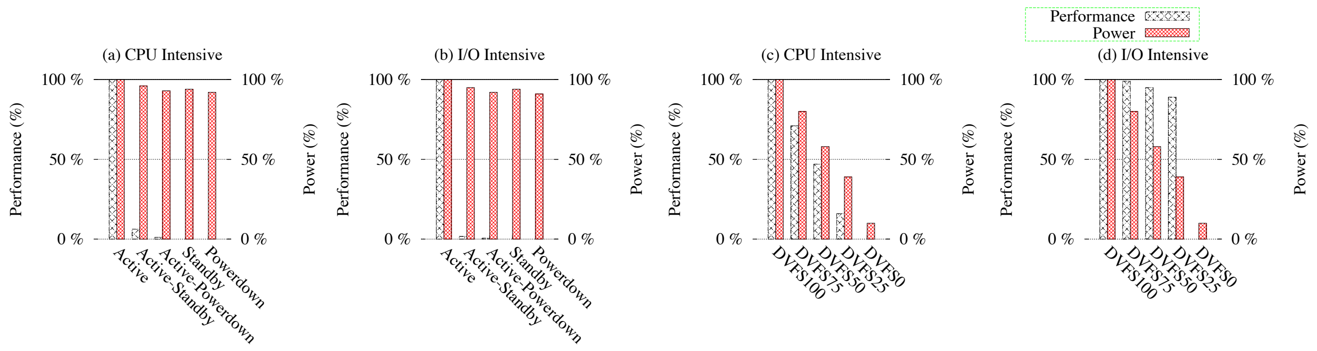


Figure 1: Performance (throughput) of a 100GB database system under low power states of the memory (Fig.a and Fig.b) and the CPU (Fig.c and Fig.d). All data are normalized to the normal scenario with active memory and CPU at 100% frequency. Subfigures are labeled by the workload type used in the test. DVFS0 is the CPU C-state, in which the system is in halt and there is no observed DBMS throughput.

Fig.1(c) and Fig.1(d) illustrate the results of many DBMS workloads running in different CPU power states. One observation is that, in both I/O intensive and CPU intensive queries, the active power cost monotonically decreases with the CPU frequency. This is in conformity with results reported in [14, 23]. The DBMS throughput, on the other hand, shows the same behavior. Such observations imply that CPU frequency and system performance are positively related and this gives us confidence in building an approximated linear system model between performance and power consumption. Nevertheless, comparing Fig.1(c) and Fig.1(d), the DBMS sensitivity[3] is different in CPU intensive and I/O intensive workloads. Apparently, one could harvest more power savings from I/O intensive queries without affecting their performance much.

Fig.1(c) and Fig.1(d) also demonstrate system reaction to the CPU C-state (DVFS0) in terms of power and performance. When the CPU is set to the C-state, the whole system is in the halt state. We did not observe any DBMS throughput although the active power consumption is low. At the same time, the delay of transiting in/out of the CPU C-state is so large that it jeopardizes the normal query execution in the DBMS, and leads to uncorrect query results. Thus, we do not implement the CPU C-state in PAT for power saving purposes but evaluate it in a simulation in our tech report [29].

The above experimental results show that CPU DVFS technique is a good candidate for the control actuator. Next we further explore the insight from results of Fig.1(c) and Fig.1(d).

CPU power states, the DBMS throughput and workload statistics: Fig.2(a), again, demonstrates the fact that the active power consumption is linearly related to the relative DVFS level. The power and the performance

[3] The sensitivity is defined as the change of performance in response to CPU frequency changes, as $\left(\frac{\text{throughput}}{\text{CPU frequency}}\right)$.

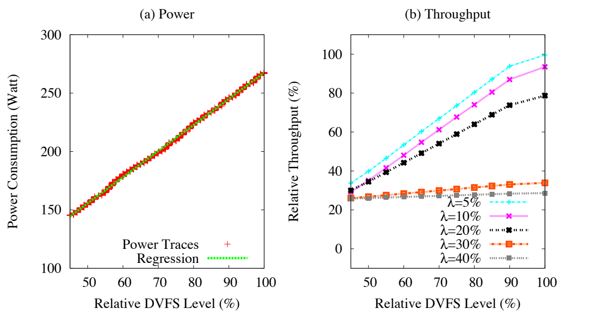


Figure 2: The impact of the CPU frequency (i.e., DVFS levels) on the power consumption (a) and the DBMS throughput (b). The five workloads in (b) differ by their λ – ratios of I/O intensive queries to the overall workload size. All data in (b) are normalized to the largest throughput of the workload with $\lambda = 5\%$ at the maximum CPU frequency.

data in Fig.2(b) are recorded from experiments running DBMS workloads with different statistics (i.e., the fraction of queries that are I/O intensive λ). An important observation from Fig.2(b) is that, there exists a linear relationship ($R^2 = 0.9633$) between throughput and CPU frequency for all DBMS workloads when λ is fixed. Therefore, we use the following linear model to describe the relationship between database throughput and CPU frequency,

$$r = A\lambda f + B \qquad (1)$$

Where r is the DBMS throughput, f is the CPU frequency, and A, B are model coefficients.

Among all the workload characteristics, we found that the ratio of I/O-intensive queries λ is the major factor that affects the sensitivity, as shown in Fig.2(b). Our explanation is that, in our platform, Linux system uses Round-Robin as the process scheduling algorithm. Therefore, the more queries are bounded by I/O, the larger chance that those processes will skip their CPU time slices, thus keeping the CPU idle. As a result, a high

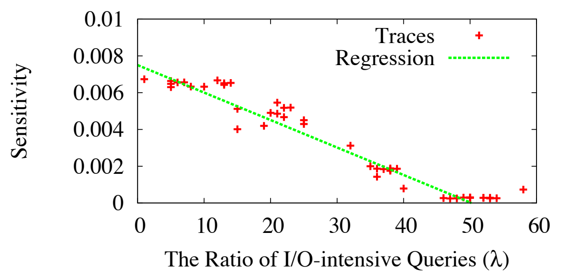


Figure 3: The relationship between workload's frequency-to-throughput ratio and the percentage of I/O-intensive queries in the workload (λ).

λ makes the system less sensitive to the CPU frequency changes. When λ is larger enough, as the grey line in Fig.2(b), the database performance has little change with the CPU frequency, where we could harvest the most energy savings. This sensitivity of the DBMS performance to the CPU frequency changes is measured as the slopes of all the throughput-DVFS lines in Fig.2(b).

Fig.3 shows that the relationship between λ and the sensitivity is also linear (with a goodness-of-fit $R^2 = 95\%$). Since λ is essential in our throughput control, it is necessary to estimate the value of λ to identify the workload at runtime. Note here, when the value of λ increases from 20% to 30% in Fig.2(b), the DBMS throughput drops heavily (50%) at the highest DVFS level. There is a value between 20% to 30%, we called it β, that defines an infection point. When the system crosses this point ($\lambda > \beta$), it will enter an I/O busy waiting state. This state is a "Limbo" that we are trying to avoid in our experiment. The value of β is a relative static number for any given systems. It can be found during the system identification process. In our experimental database system, the value of β is found to be 32%.[4]

3 The Framework of PAT

The control framework PAT is illustrated in Fig.4. The main components of PAT form a feedback control loop (indicated by the red arrow in Fig.4), including the PI controller (Controller), the throughput monitor of the DBMS (Plant),[5] and the CPU power state modulator (Actuator). The goal of the control loop is **to maintain the DBMS throughput at the set point R_s and minimize the power cost**. Specifically, the following steps are invoked in each control period,

1. The throughput monitor measures system throughput $r(i-1)$ in the last period. The *control error* is computed as $\Delta r(i) = R_s - r(i-1)$;

2. The controller receives the control error Δr and the workload statistic factor λ. Based on these values, it computes the *control signal* $f(i)$;

3. The CPU power state modulator receives the control output f, to calculate the new DVFS level and apply it in the CPU.

4. Exception: when $\lambda > \beta$ or the DVFS level is highest but the detected throughput still fails to meet the set point R_s, the CPU modulator will set the DVFS level to the active lowest state to save power for a short period of time t.

[4] The value of β needs to be calibrated when PAT is applied to a different system environment

[5] Note that the monitor itself is not the plant, the DBMS is.

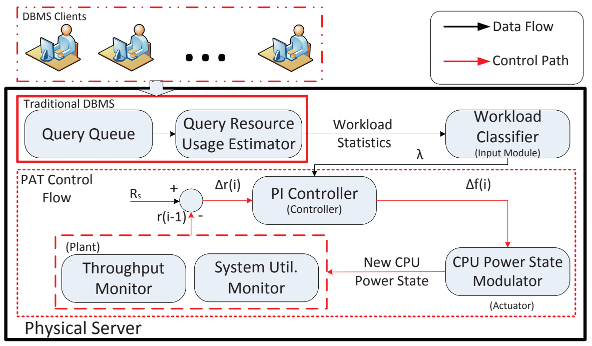


Figure 4: The power-aware throughput control architecture. The names in parentheses are given following control terminology.

Note here, the scale-down duration t is shall be smaller than control period T (introduced in Section 5), and the transaction time of different power state is at least one order of magnitude smaller than t.

3.1 Control Components

Before discussing more details about the two major components – the fuzzy workload classifier (FWC) and the PI controller, we briefly introduce the implementation of other components of PAT first.

CPU Power State Modulator: PAT uses Intel's SpeedStep technique (10ms overhead) to tune the CPU DVFS level. An interesting issue is that the Intel Xeon CPU E5645 used in our platform (as well as many other DVFS-enabled CPUs) only supports several discrete CPU frequency levels. However, PAT needs to set a value of the DVFS level within a normalized continuous range [0 – 100]. Therefore, the task of the modulator is to approximate the desired value using a combination of the supported discrete frequency levels. For example, to get 2.23 GHz CPU frequency during one control period, the modulator would output pseudo frequency signal sequence as {2.67, 2, 2, 2.67, 2, 2} to emulate the average CPU frequency as 2.23 GHz. To realize such idea, we implemented a first-order delta-sigma modulator in the system, which is commonly used in analog-to-digital signal conversion [12].

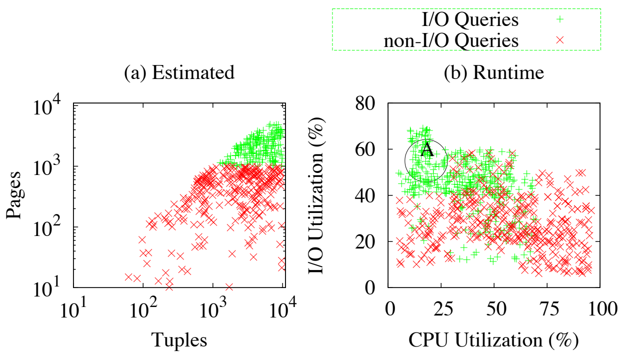


Figure 5: The estimated resource demand (Fig.(a)) and the real resource usage (Fig.(b)) of 500 different queries. Each node in the figure stands for one query.

Throughput Monitor: the throughput monitor is implemented as a daemon program that collects the number of finished queries in each control period. The monitor traces every exit signal from DBMS threads and records the sum in period *i*. The monitor maintains throughput data from the past *n* periods.

System Utilization Monitor: the system utilization monitor is a program that records system status (e.g., CPU utilization) when PAT is active. We use the collected data for the performance analysis.

Query Resource Consumption Estimator: the resource estimator is a tool that retrieves the run-time query estimation information from the query optimizer of the DBMS. Based on such information, FWC could define the query type and update the corresponding parameter λ. Note that the original resource estimator in the DBMS could be highly unreliable. We calibrate this estimator before using it for estimation. The detailed work can be found in the tech report [29].

4 Fuzzy Workload Classifier

As we learned from the observations in Section 2, the key factor in the workload statistics used in the system model is the percentage of I/O intensive queries in the current running workload, namely λ. Thus, to successfully build the system model in order to control a composite workload, the model shall update the value of λ on-the-fly. To solve this problem, we need to classify queries based on its I/O intensity.

4.1 Main Challenge

Fig.5(a) shows the classification results of 500 queries based on a static threshold – 1,000 demanding pages, which is the size as the L3 cache in the server. Each query is labeled as I/O-intensive if the I/O cost is more than 1,000 pages (green node) or non-I/O intensive (red node), otherwise. However, such a rule-based method fails when the resource estimation given by the query optimizer is not accurate enough to reflect the actual requested resource at runtime. Fig.5(b) shows the real resource usage of the same set of queries. The results show only a part of identified I/O queries are real I/O-intensive queries (e.g., those nodes in circle A in Fig.5(b)). The above empirical results show that simple rule based classification fails at obtaining the accurate λ value.

We propose a classification approach based on fuzzy set theory to solve our problem. Fuzzy-based methods are particularly suitable for systems with complex behaviors. They are designed to handle an unpredictable environment with limited number of rules to reach sufficient accuracy [25, 21]. Our FWC collects workload statistics and creates fuzzy rules to identify runtime resource consumption patterns of queries in the workload.

4.2 Fuzzy Classifier Design

In FWC, Sugeno-type fuzzy rules [21] are generated from the clustered data for modeling database workloads. The input for the FWC is the resource demand of the incoming query and the output is the aggregated estimation of runtime resource utilization. For the i^{th} query, its resource demand vector is denoted as $[d^i_{CPU}, d^i_{I/O}]^T$ and the estimated CPU and I/O utilization as $[u^i_{CPU}, u^i_{I/O}]^T$. The number of fuzzy rules shall be the same as the number of clusters in the estimated resource demand map [8]. In our case, there are two clusters: I/O-intensive and non-I/O-intensive. The member functions of the fuzzy rules are linear functions generated via rigorous mathematical tools from Matlab [13]. The rule base is constructed as follows:

R_j: IF $[d^i_{CPU}, d^i_{I/O}]^T \in$ cluster X_j, THEN $[u^i_{CPU}, u^i_{I/O}]^T = M_j[d^i_{CPU}, d^i_{I/O}]^T + N_j$

where X_j is the cluster determined by clustering technique, M_j and N_j are parameters from the fuzzy set associated membership functions obtained from the learning process. The symbol $\in$ stands for the distance between the node and the center of cluster X_j. The procedure of workload classification are as follows:

1. *Evaluation*: compute the appropriate fuzzy rule output $[u^i_{CPU}, u^i_{I/O}]^T$ based on the input resource demand vector $[d_{CPU}, d_{I/O}]^T$ using the corresponding membership functions M_j and N_j;

2. *Implication calculation*: obtain implication p_j of each fuzzy set R_j and calculate the confidence t_j that the query belongs to fuzzy set R_j based on the implication weight over all $\frac{\Sigma(p_j)-p_j}{\Sigma(p_j)}$;

3. *Aggregation result*: the output of all fuzzy rules are aggregated and inversely translated into the av-

erage utilization vector $[\sum_j t_j u^i_{CPU}, \sum_j t_j u^i_{I/O}]^T$ from all rules with confidence t_j.

Although there still exist errors from workload classification, those errors are bounded and smaller than the acceptable maximum tolerance (overshoot) in the controller design. Due to the page limit, we put a detailed analysis and examples of FWC in our tech report [29]. The accuracy of FWC is evaluated in Section 6.

5 Throughput Controller Design

5.1 System Modeling

Building an accurate mathematical model of the system to be controlled is of great importance to the entire control loop design. We build the model of the DBMS throughput and the power consumption based on observations in Section 2. Let us denote the length of the control period as T and the throughput within the i^{th} period as $r(i)$. Given $r(i)$, our control goal is to guarantee that the DBMS throughput r could be converged to the set point R_s after a finite number of control periods (*settling time*). Note here, for better establishing the model we scale those two values into percentage. Thus, Δr and f are now the relative control error and the related frequency setting, respectively. For example, $f = 100\%$ means that CPU is running at its highest frequency. In the experiment, the minimum available frequency is 40% of the maximum frequency.

Here we update the system model in Eq. (1) as:

$$\Delta r(i) = \lambda A f(i) + B \tag{2}$$

For the convenience of the control analysis, Eq. (2) is transformed in the z-domain as:

$$R(z) = \lambda A F(z) \tag{3}$$

where $R(z), F(z)$ are the z-transform of signal $\Delta r(i), f(i)$, respectively. Thus, the system transfer function of the DBMS throughput to the frequency change in Fig.2 is:

$$G(z) = \frac{R(z)}{F(z)} = \lambda A \tag{4}$$

We test the system with sinusoidal inputs in Fig.6. Fig.6 demonstrates that our model is sufficiently close to the actual system with $R^2 = 0.9152$.

5.2 Controller Design

The goal of the controller design is to meet the following goals:

- *stability*, the throughput shall settle into a bounded range in response to a bounded reference input;

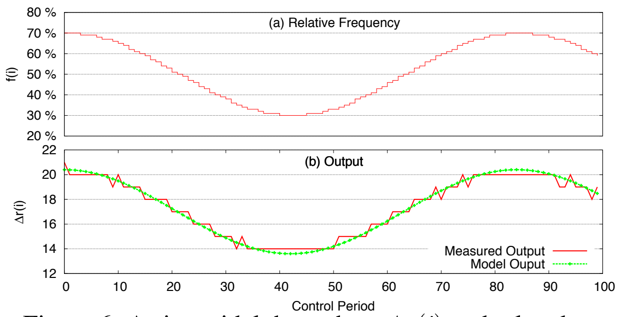


Figure 6: A sinusoidal throughput $\Delta r(i)$ and related control signal $f(i)$.

- *zero steady state error*, when the system enters the steady state, the throughput shall settle to the set point with zero errors; and
- *short settling time*, the system shall settle to the set point before the specified deadline.

Based on the control theory, we design a *Proportional-Integral* (PI) control that has been widely adopted in industry control systems. We select the PI controller for its nice property of the zero-state-error and its fast response [4, 7]. The PI controller can also provide robust control performance despite modeling error and input/output disturbances. It has the following form in the discrete time domain:

$$f(i) = k_P \Delta r(i) + k_I \sum_{1}^{i} (\Delta r(j)) \tag{5}$$

where $\Delta r(i)$ is the control error at i^{th} period. $f(i)$ is the frequency offset. k_I and k_P are control parameters. Those parameters can be analytically chosen to guarantee the system stability and zero steady-state error. From Eq. (5), we have the controller transform function in the z-domain as:

$$C(z) = \frac{z(k_I + k_P) - k_P}{z - 1} \tag{6}$$

Overall, the transfer function $\mathbf{F}(z) = G(z)C(z)$ is,

$$\mathbf{F}(z) = \frac{\lambda A k_p (z-1) + \lambda A k_I z}{(1 + \lambda A(k_I + k_P))z - (\lambda A k_P + 1)} \tag{7}$$

We use the Root-Locus method [7] to design the control coefficients k_I and k_P to guarantee stability and zero steady-state error. The poles of the transfer function are $-0.26 \pm 0.8i$. As both eigenvalues are inside one unit circle, the closed-loop system in our experiments is stable [4]. The values of the system model parameters in Eq. (2) are $A = 4.329$ and $B = 24.329$, based on our characterization study. λ is provided at runtime by FWC. Based on the result of control analysis, control parameters $k_I = 0.5$ and $k_P = 1.06$. More details of the control analysis are in the tech report [29].

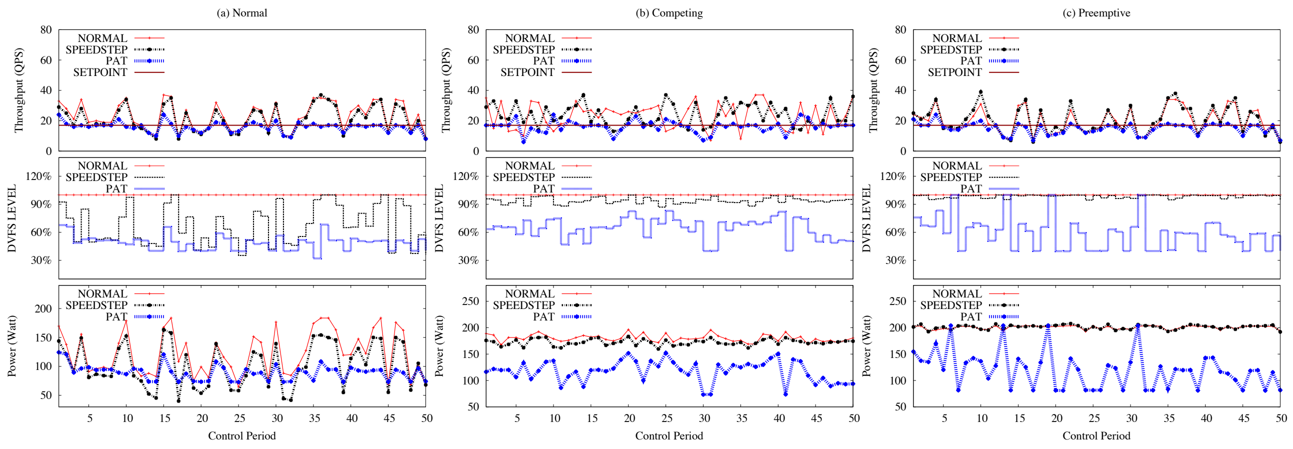


Figure 7: A snapshot of the database throughput, the DVFS control signal and the active power consumption of NORMAL, SPEEDSTEP and PAT in three different system settings.

6 Performance Evaluation

6.1 Experimental Setup

Our test bed contains an open-source database PostgreSQL (version 8.3.18) running under Redhat 5 (kernel version 3.0.0). The data server is a DELL PowerEdge R710 with 12-core Intel Xeon E5645. A client feeds the server with a typical database workload generated from TPC tools [22] and SDSS traces [20]. We use a WattsUp power meter ($\pm 1.5\%$ error, 1 Hz sampling frequency [1]) to measure the power consumption.

We have designed several baselines for evaluation.

1) *NORMAL and TRADITION*: those two baselines are common Advanced Configuration and Power Interfaces (ACPIs) in modern servers. *NORMAL* is when system runs with the maximum CPU frequency and *TRADITION* sets a static CPU frequency based the offline workload analysis.

2) *SPEEDSTEP, HEURISTIC and SCTRL*: *SPEEDSTEP* is the ACPI policy in BIOS that tunes the CPU frequency according to the system load. *HEURISTIC* is an *ad hoc* control solution with the DBMS performance set point R_s. *SCTRL* is an OS-level feedback control solution with the DBMS performance set point R_s. Comparing with PAT, it contains the basic control loop with throughput monitor to detect the DBMS throughput except the FWC and any internal parts of the DBMS in Fig.4. Note that, when those control solutions control the CPU frequency, other power management policies are turned off.

6.2 Performance of PAT

To study the impact of PAT on performance and energy savings, we have designed three scenarios from daily DBMS operations. 1) *Ideal environment*: the database process is the only user of all the computational resources; 2) *Competing environment*, there exists a set of pure CPU intensive programs in the system competing for the CPU resource. 3) *Preemptive environment*, there exists a set of high-priority (OS-level) processes which randomly occupies the CPU resource assigned for database processes. Fig.7 shows the database throughput, the DVFS control signal, and the active power consumption of the system using NORMAL, Speedup and PAT ($R_s = 17$QPS) in above scenarios in 50 control periods.

In Fig.7(a), SPEEDSTEP and PAT provide significantly larger energy savings than NORMAL does. Comparing with SPEEDSTEP, PAT controls the throughput performance strictly to the setpoint, and the maximum overshoot (throughput exceeding the set point) is much smaller.

In the competing scenario in Fig.7(b), the database throughput is greatly affected by the competing CPU-intensive processes, which are injected into the system follows a Poisson distribution. The noise from resource competition between database processes and CPU-intensive processes hurts the control performance of PAT. However, PAT could tolerate such noise and control the throughput back to the setpoint within 3 periods. On the other hand, because SPEEDSTEP controls the CPU frequency based on the total system utilization, it usually sets the DVFS level near the highest level.

Fig.7(c) demonstrates the results in the preemptive scenario. The preemptive behavior of system processes leads to a low DBMS throughput due to the interrupt and resource occupation. It is often the case when the actuator fails to handle the overshoot exceeding its control limit. PAT treats this case as the exception and tunes down the CPU frequency to save more energy, such as the 6th, 13th, 18th, etc. period in Fig.7(c).

Overall, PAT saves up to 51.3% of the energy con-

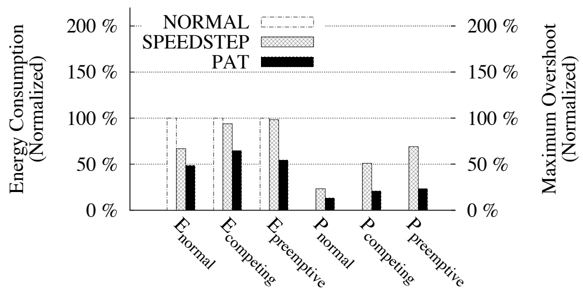


Figure 8: The normalized energy consumption and the performance overshoot of NORMAL, SPEEDSTEP and PAT in three scenarios. The energy cost is normalized to the data of NORMAL.

sumption (15% more than the SPEEDSTEP), comparing with NORMAL in the ideal environment, shown in Fig.8. Its maximum performance overshoot[6] is less than half of the overshoot in SPEEDSTEP, especially in the competing and preemptive scenarios. This is because SPEEDSTEP does not take the DBMS performance as the control goal and the system dynamic gives more error in the last two scenarios. Here we show the advantage of PAT by comparing with ACPI baselines. To further study the performance and the robustness of PAT, we test it with other control baselines to control the DBMS throughput.

6.3 Control Performance Comparison

Fig.9 is the snapshot of the database throughput, DVFS setup and the power consumption of four controllers in the ideal system environment.

TRADITION cannot control the throughput to the set point because the workload does not always follow the pattern in the offline analysis. This is a typical problem of open control. First, finding a good static DVFS for one workload in one system scenario needs extensive experimental work and complex learning processes. Second, it could easily fail under workload variations.

HEURISTIC gives a relatively better control performance, comparing with the SPEEDSTEP. However, when facing an ever-changing workload, HEURISTIC fails to commit to a steady state in an acceptable time. For example, data in control period 20 to 30 in Fig.9(a) show how HEURISTIC fails to handle the “M” shape throughput pattern. While SCTRL and PAT could both commit to the setpoint in 4 periods, the tuning of HEURISTIC oscillates in many steps, which results in less energy savings. Solving the problem will eventually leads to the same feedback controller design in PAT.

SCTRL treats the DBMS processing as a black box. It settles to the setpoint faster than HEURISTIC. However, when all DBMS processes are in I/O busy waiting,

[6] the performance overshoot is measured by P_{max}/R_s, where P_{max} is the maximum performance and R_s is the set point

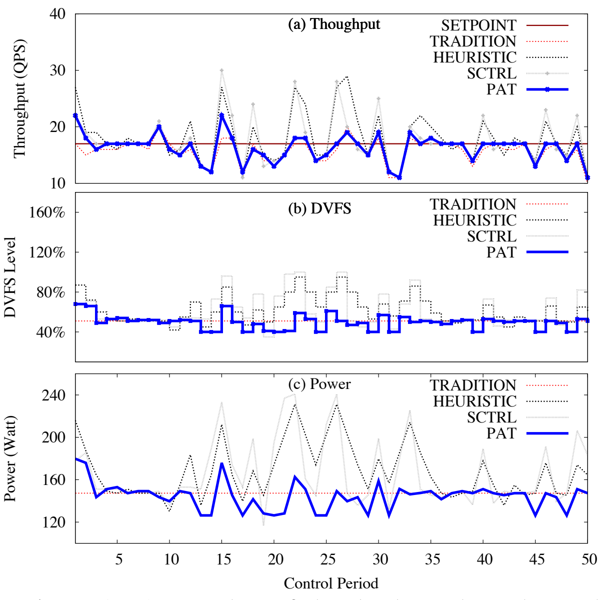


Figure 9: A snapshot of the database throughput, the DVFS control signal and the active power consumption of the four control solutions.

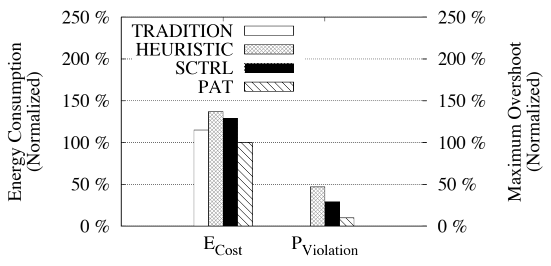


Figure 10: The normalized energy consumption and the performance overshoot of four control solutions.

SCTRL would uselessly set the DVFS level to the maximum, and waste energy.

We conduct the overall performance evaluation of the five control technique in Fig.10. While PAT achieved the 20% more energy savings than that of TRADITION, HEURISTIC and SCTRL only got 56% and 74% of energy savings achieved by PAT because of the failure to commit steady state (HEURISTIC) and the unnecessary setting of the highest DVFS level (SCTRL). Comparing the performance violation, PAT has the smallest maximum overshoot than the other two control methods.

6.4 The performance of FWC

Our fuzzy workload classifier provides a high prediction accuracy of the query resource consumption pattern. The classification result of the tested workload above is shown in Fig.11. The accuracy is above 90% for the two testing traces. FWC provides high accuracy in the system model for controller design in PAT. Theoretically, PAT could tolerate up to 45% overshoot from model and environment based on the controller design. As shown

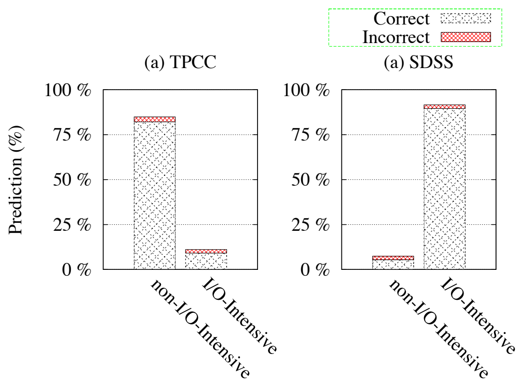


Figure 11: The prediction result of the query processing pattern using FWC.

in Fig.7 and Fig.9, the biggest difference of the workload trends is almost 40% but PAT could still control the throughput to the set point in a few periods.

7 Related Work

Reduction of energy consumption has become an active topic in the DBMS community. Harizopoulos et al. [6] and Graefe [5] introduce a new paradigm of DBMS design concerning energy efficiency. Recent work [30] shows that there exists energy-efficient query plans in DBMS. Another work by Harizopoulos [23] suggests that most energy efficient plans come from DBMS running in the active low power mode. Poess and Nambiar [19] examine multiple storage components in the system for energy saving potentials. Lang et al. [11] claim that it is worthwhile to scale hardware performance to control in DBMS's query processing in the distributed environment. In contrast to their work, we argue that applying hardware scaling technique to DBMS design for energy-saving purposes is not a trivial task and propose a systematic solution that relies on rigorous control loop design. As compared to heuristics-based strategies, our solution provides analytical assurance of control accuracy and system stability.

Application of mathematical control theory has been conducted in several topics in the DBMS area. Tu et al. [24] introduce this technique to handle load shedding in data stream systems by using a classic P feedback controller that successfully avoids noticeable streaming tuple delays with lower data loss. Kang et al. [9] create Chronos by applying a similar feedback model in controlling number of transactions to a baseline. Our paper, unlike those two, is one of the first attempts targeting at energy savings while preserving performance in database systems. It is inspired by the fact that most database servers are running in relatively low utilization – energy proportionality can be achieved if we make database run under low power modes of hardware.

Recently, feedback control theory has been successfully applied to energy efficient control for data center servers at the system and hardware levels [2, 26, 17]. Existing solutions of power and performance control for enterprise servers attempt to tackle the problem in two separate ways. Performance-oriented control solutions focus on altering power to meet the system-level performance budget while reducing power consumption in a best effort manner [16]. However, those solutions do not have any explicit internal information from software, such as DBMS. As a result, there could be undesirable performance degradation. In the other way, power-oriented control solutions treat power as the first-class control target and maximize the performance within the power budget [2, 26, 28]. In DBMS, its throughput could not be maximized by control at the system level because the resource are evenly distributed (Round Robin scheduling in Linux). Thus, we need to build the control loop by taking DBMS statistics into consideration.

8 Conclusion and Future Work

The contradictory requirements of high performance and low energy consumption have attracted a lot of talents working on database system design. The low-power modes of hardware provide opportunities for power saving with predictable performance degradation. In this paper, we tackle the problem of maximizing energy savings under a user-specified performance bound in database systems. We argue that such a problem is non-trivial due to the dynamics in database workloads and environment. Therefore, based on the results of our evaluation, traditional offline analysis and heuristic solutions are not effective. We propose our solution as a feedback control framework based on system characteristics. Unlike heuristic-based adaptive solutions widely used in database tuning, PAT provides performance guarantees over the power control on hardware. We implement PAT with the PostgreSQL engine and the empirical results demonstrate that PAT can achieve high energy efficiency with small violation of SLA. One immediate future work is to consider the performance bound of individual queries using DVFS as the global control actuator.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by the US National Science Foundation (NSF) grants IIS-1117699, IIS-1156435, and CSR-1143607.

References

[1] Watts up power meter. `http://goo.gl/AI7so`.

[2] M. Chen, X. Wang, R. Gunasekaran, H. Qi, and M. Shankar. Control-based real-time metadata matching for information dissemination. In *RTCSA*, pages 133–142, 2008.

[3] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: active low-power modes for main memory. *SIGPLAN Not.*, 46(3):225–238, Mar. 2011.

[4] G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, 1990.

[5] G. Graefe. Database servers tailored to improve energy efficiency. In *SETMDM*, pages 24–28, 2008.

[6] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy efficiency: The new holy grail of data management systems research. In *CIDR*, 2009.

[7] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31.

[9] K.-D. Kang, J. Oh, and S. H. Son. Chronos: Feedback control of a real database system performance. In *RTSS*, pages 267–276, 2007.

[10] M. Kunjir, P. K. Birwa, and J. R. Haritsa. Peak power plays in database engines. In *Proc. of EDBT*, EDBT '12, pages 444–455. ACM, 2012.

[11] W. Lang, R. Kandhan, and J. M. Patel. Rethinking query processing for energy efficiency: Slowing down to win the race. *IEEE Data Eng. Bull.*, 34(1):12–23, 2011.

[12] C. Lefurgy, X. Wang, and M. Allen-Ware. Server-level power control. In *ICAC*, page 4, 2007.

[13] MATLAB. *version 7.13.0 (R2011b)*. The MathWorks Inc., Natick, Massachusetts, 2011.

[14] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, pages 319–330, New York, NY, USA, 2011. ACM.

[15] D. Meisner and T. F. Wenisch. Dreamweaver: architectural support for deep sleep. In *ASPLOS*, pages 313–324, 2012.

[16] R. G. Melhem, D. Mossé, and E. N. Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. Computers*, 53(2):217–231, 2004.

[17] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, pages 13–26, 2009.

[18] M. Poess and R. O. Nambiar. Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results. *PVLDB*, 1(2):1229–1240, 2008.

[19] M. Poess and R. O. Nambiar. Tuning servers, storage and database for energy efficient data warehouses. In *ICDE*, pages 1006–1017, 2010.

[20] Sloan Digital Sky Survey. `http://goo.gl/hOVDP`.

[21] T. Takagi and M. Sugeno. Fuzzy identification of systems and its applications to modeling and control. *Ieee Transactions On Systems Man And Cybernetics*, 15(1):116–132, 1985.

[22] Transaction Processing Performance Council. `http://www.tpc.org`.

[23] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *SIGMOD*, pages 231–242, 2010.

[24] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, pages 787–798, 2006.

[25] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. B. Fortes. Fuzzy modeling based resource management for virtualized database systems. In *MASCOTS*, pages 32–42, 2011.

[26] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. Ship: Scalable hierarchical power control for large-scale data centers. In *PACT*, pages 91–100, 2009.

[27] Y. Wang, X. Wang, M. Chen, and X. Zhu. Partic: Power-aware response time control for virtualized web servers. *IEEE Trans. Parallel Distrib. Syst.*, pages 323–336, 2011.

[28] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh, and D. W. Clark. Formal control techniques for power-performance management. *IEEE Micro*, 25(5):52–62, 2005.

[29] Z. Xu. Model Evaluation of PAT, a comprehensive study. Technical report. `http://goo.gl/Cd2sN` (Link is shortened).

[30] Z. Xu, Y.-C. Tu, and X. Wang. Exploring power-performance tradeoffs in database systems. In *ICDE*, pages 485–496, 2010.

Wireless Inference-based Notification (WIN) without Packet Decoding

Kevin Chen and H. T. Kung
School of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138, USA

Abstract

We consider ultra-energy-efficient wireless transmission of notifications in sensor networks. We argue that the usual practice where a receiver decodes packets sent by a remote node to acquire its state or message is suboptimal in energy use. We propose an alternative approach where a receiver first (1) performs physical-layer matched filtering on arrived packets without actually decoding them at the link layer or higher layer, and then (2) based on the matching results infers the sender's state or message from the time-series pattern of packet arrivals. We show that hierarchical multi-layer inference can be effective for this purpose in coping with channel noise. Because packets are not required to be decodable by the receiver, the sender can reach a farther receiver without increasing the transmit power or, equivalently, a receiver at the same distance with a lower transmit power. We call our scheme Wireless Inference-based Notification (WIN) without Packet Decoding. We demonstrate by analysis and simulation that WIN allows a sender to multiply its notification distance. We show how senders can realize these energy-efficiency benefits with unchanged system and protocols; only receivers, which normally are larger systems than senders and have ample computing and power resources for WIN-related processing.

1. Introduction

We consider a common sensor network scenario where remote senders, such as sensors, transmit notifications about event detected as well as their operational conditions (e.g., device operating normally, and remaining battery power) to some designated receivers over wireless channels. In such a scenario, it is often desirable that nodes draw only a small amount of power in transmitting such notifications. This would allow transmitters to survive for a long time like years even operating on a small coin battery, in applications such as industrial monitoring and home automation.

Under a conventional approach (e.g., [1]), we will adopt a low-power wireless network, e.g., Bluetooth or ZigBee, to send notifications. A sender will periodically transmit *normal packets* to report that it is in a *normal state*, and start transmitting *event packets* when it enters an *event state* upon noticing events of interest. A receiver will decode each received packet to determine if it is a normal or event packet, and in the latter case, may also examine packet payload to obtain further information about the event. In real-world applications, we expect that the bulk of the transmission is for normal packets and transmission of event packets is relatively infrequent. This means that it is especially important for the sender to minimize transmission energy for normal packets, while being able to quickly alert the receiver when events of interest occur.

We argue that for many sensor applications this conventional approach is suboptimal in terms of energy use. For example, there is no need for the sender to transmit at a relatively high transmit power to ensure all these normal packets transmitted can be decoded by the receiver, if the time series of packet arrivals can already reveal that the sender is in the normal state. Upon noticing events of interest a sender merely need to seek attention from the receiver about the new situation. To this end, the sender can just transmit packets with a different pattern in time series. The receiver can then use a robust inference method to classify the sender being in a normal or event state based on patterns in the time series of packet arrivals, without having to decode packets.

In this paper we explore such inference-based approaches where no packet decoding is required. This would enable the receiver to operate at a lower signal-to-noise ratio (SNR), and, in turn, allow the sender to reach receiver at the same distance with lower transmit power or, equivalently, farther receivers with the same transmit power.

A key issue with such approaches is their accuracy in classifying the current state of the sender in low SNR situations when the receiver is distance away, and/or the wireless chancel is noisy. We show in this paper how a two-layer hierarchical inference can be effective

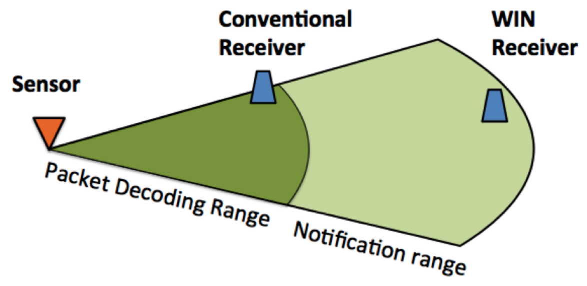


Figure 1. The WIN receiver can receive notification from

in providing robust and reliable classification based on the packet arrival patterns, even when some packets may have distorted symbols or may be completely lost. We call our approach Wireless Inference-based Notification without Packet Decoding, or for short, WIN.

2. Observations on Wireless Sensor Network Communications

We consider a wireless sensor network where the messages being sent are relatively stationary. For example: fire alarm sensors in a building or thermal sensors in an exhibition area routinely report their status through a wireless channel. In these scenarios where repeated traffic patterns are expected, a receiver can come to learn the probable patterns of incoming packets. As described below our approach has several advantages over the conventional packet decoding method.

2.1. Posterior Probability Estimation of Codewords

Conventional wireless communication assumes no prior knowledge of what we expect to receive. Therefore all codewords are considered equally likely. Let c_i be the message being sent and x be the received signal. Decoding algorithm makes decisions by maximum likelihood estimation (MLE):

$$p(c_i|x) = p(x|c_i)p(c_i) \propto p(x|c_i)$$

given that the prior $p(c_i)$ is assumed to be a constant. It is then the interest of channeling coding to design codes with efficient decoding algorithm that approximates MLE.

However, note that optimal decisions should be based on the real posterior probability $p(x|c_i)p(c_i)$. While this may be difficult to implement in general, simple solutions would suffice in the case were only a few codewords are likely to occur. In a stable sensor network environment, this may be a more fitting assumption than uniform prior. In this paper, we consider the case where c_i is restricted to $\{inactive, normal, event\}$, and demonstrate significant gains for utilizing this prior knowledge.

2.2. Multi-layer Inference

When a single packet provides insufficient evidence about the state of sender, the receiver can wait for other incoming packets for better inference results. A receiver can be a lot more powerful if is allowed to accumulate information overtime time. In WIN, we model packet arrival patterns in addition to just packet patterns, so that even undecodable packets can be useful.

2.3. Classification with Respect to False Negative and False Positive

Not all errors are equal. Sometimes it is safe to misclassify normal state as an event while mistaking event as normal could lead to more severe consequences. We can make decisions according to the posterior probability with respect to false positive/negative rates required by application. While this may not be possible for conventional wireless communication due to the complexity of applications, it is doable for many sensor scenarios and should not be overlooked.

3. Overview of the WIN Approach and Comparison with Conventional Methods

We describe the conventional approach of transmitting notifications, and then describe at a high level how our proposed WIN approach can accomplish the same task with lower energy consumption.

Conventional methods include wireless networks designed for energy-constrained applications, such as Bluetooth LE, ANT+ or ZigBee [2, 3, 4]. While hardware and protocols of these networks have been optimized for low-energy senders, they are still based on the conventional network-layering abstraction. In particular, packets must be decoded at the link or a higher layer in order to reveal packet load that contains notification messages. To be specific, in the rest of the paper, we will use Bluetooth LE [9] as our comparison target.

As depicted in Figure , under the conventional approach a sender periodically transmits normal packets (black) to a receiver to report that the sender is alive and it is in a normal state. Upon noticing events of interest, the sender enters the event state and starts transmitting event packets (red). The receiver will attempt to decode every received packet to determine the state of the sender.

Under a corresponding WIN approach, the sender in the normal state will periodically transmit normal packet like in the conventional approach. When the sender enters the event state, it will transmit event packets pe-

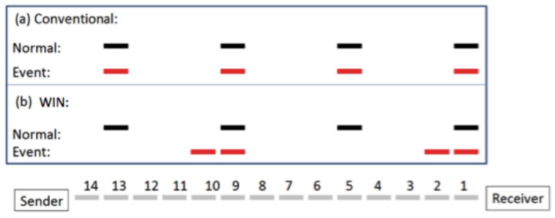


Figure 2. Conventional approach vs. WIN. Time slots labeled by time are shown at the bottom. Solid bars denote normal (black) and event (red) packets transmitted at various time slots.

riodically under a different arrangement about the length of packet burst or gap. Figure depict of an example of such a WIN scheme based on the following time series of packet transmissions:

Normal state: burst =1 and gap = 3
Event state: burst = 2 and gap = 6

Note that in supporting WIN, a conventional sender does not need to change its protocol stack; all it needs to do is to change packet transmission patterns during the event state. Thus existing sensor transmission systems are readily useable. This is an advantage over other approaches that also exploit physical layer signal properties [5].

The receiver employs physical-layer matched filters to determine whether each time slot has an arriving packet. Based on the matching results from multiple time slots, the receiver uses inference methods to infer the state of the sender (see Sections 4). By making use of aggregated matching results from multiple time slots and leveraging the designed-in separation between the time series of packet transmissions for the normal vs. event state, as we will show later, a WIN receiver can operate at a lower SNR. As a result, a distant receiver may still be able to infer the state of the sender even it cannot decode normal or event packets. This is illustrated in **Figure 1**. When a receiver determines that the sender is in the event state, should the receiver happen to be mobile, it could move itself closer to the sender to decode the event packet and learn about the event. Alternatively, the receiver may dispatch other agents for the task.

4. Inference Methods Used by WIN

WIN infers the state of the sender from physical layer measurements on arrived packets. The receiver matches arriving signals against a dictionary of patterns corresponding to the sender's states. Consider, for example, the scenario displayed in Figure , where the sender

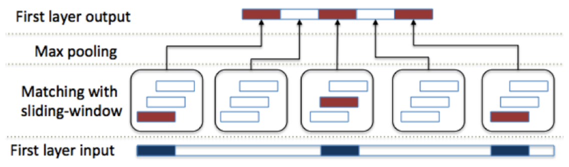


Figure 3. When detecting packets in the first layer, we use max-pooling with a sliding-window to address variations in packet delay due to multipath.

transmits one packet every four slots in the normal state, and 2 back-to-back packets every eight slots in the event state. No packets will be sent when sender is inactive. Hereafter, we refer these time slots as subintervals.

We use a two-layer hierarchical model to infer the state of the sender. In the first layer, we perform a filtering operation matching the observed signal with t, the target packet pattern, using sliding-window across all possible locations within a subinterval. The sliding-window allows us to detect the packet even with delays variances caused by multipath [6]. We then take the max of these values and call it m_i for subinterval i. This value reflects the likelihood of the target pattern t being present in subinterval i. See Figure 3 for an illustration.

In the second layer, we match m_i with arrival pattern, and classify according to the result, m. Figure 4 depicts a simulation result on the distribution of m conditioned on the three different states, at -15dB SNR. As shown in the figure, the distribution of m is fairly close to Gaussian distribution, which can be explained by central limit theorem. The inferred state s is selected according to:

$$s = \begin{cases} inactive, & m < t_n \\ normal, & m < t_e \\ event, & m \geq t_e \end{cases}$$

where thresholds t_n and t_e are chosen to satisfy a notification false positive rate R_n and an event false positive rate R_c.

Our method is a special case of a two-layer model that computes sparse representations of input in machine learning [7]. Our problem here is simpler because we can design the dictionary and assure that the dictionary entries are well separated to increase inference accuracy.

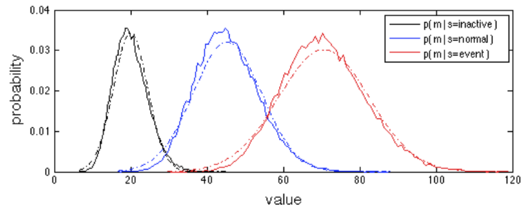


Figure 4. Probability distributions of the value of the matching metric $\boldsymbol{m}$ or state inactive, normal and event. Dotted lines are approximations with Gaussian.

5. Performance Analysis

We compare the conventional approach and WIN by computing their probabilities of successful transmission of notification. For both methods, the sender is allowed to transmit at most R packets, where each packet consists of n bits. A conventional transmission is successful if the receiver correctly decodes a packet with no CRC error. A WIN transmission is successful if the sender's state is classified correctly. For this analysis, we consider the AWGN (additive white Gaussian noise) channel with no packet delays.

Performance of Conventional Approach

A conventional method would only fail when none of the R packets pass the CRC. Thus,

$$p(fail) = \left\{1 - \left(1 - \frac{1}{2} erfc(SNR)\right)^{n}\right\}^{R}$$

where $BER = \frac{1}{2} erfc(SNR)$ for some $SNR = \frac{E_b}{N_0}$.

WIN Performance

In WIN, transmission fails if a state is misclassified. We will first find the distribution of detector z for each time slot, and then derive the distribution of the second layer detector m. Finally, we will estimate the probability of classification error by WIN.

Let t be the pattern of a packet in physical layer, and y be the sensed signal. We consider the hypothesis test on hypotheses H_0 and H_1:

$$y = \begin{cases} H_0: w \\ H_1: t + w \end{cases}$$

where $w \sim N(0, \mathbf{I})$ is noise from an AWGN channel. Then, a physical-layer detector based on matched filter can be expressed as $z = |t^H y|^2 = y^H t t^H y = yTy$ where $T = tt^H$ is a rank-1 matrix. We follow the analysis by Reed et al to compute false positive rates in detecting packets with matched filter, which gives the distribution of z as summarized in the following theorem [8]:

THEOREM 1 *If* $\mathbf{x}$ *is* $N_m(\mathbf{m}_x, \mathbf{I}_m)$ *and* $\mathbf{B}$ *is an* $m \times m$ *projection matrix of rank k then* $\mathbf{x}^H \mathbf{B} \mathbf{x}$ *has a noncentral* $\chi_k^2(\delta)$ *distribution where* $\delta = \mathbf{m}_x{}^H \mathbf{B} \mathbf{m}_x$.

By theorem 1, the distribution of z is

$$\chi_1^2(d) = e^{-z-|t|^2} \mathrm{I}_0\left(2\sqrt{z|t|^2}\right)$$

where I_0 is the modified Bessel function of the first kind. We have $d = d_0 = 0$, $d = d_1 = t^H t = |t|^2$ for the two hypotheses H_0 and H_1, respectively. The mean and variance (μ, σ^2) of z is (1,1) under H_0 and (d_1+1, 2d_1+2) under H_1. Given $SNR = \frac{E_b}{N_0}$ under unit variance Gaussian noise, we have $|t|^2 = |nSNR|^2$ where n is the number of bits per packet. Now, we have the distribution of matched filter detector z as a function of channel SNR.

Let Z_0 and Z_1 denote the random variables drawn from $p(z|H_0)$ and $p(z|H_1)$. The second layer detector m is then

$$m \sim \begin{cases} inactive: M = RZ_0 \\ normal: M = \frac{R}{2}(Z_0 + Z_1) \\ event: M = RZ_1 \end{cases}$$

where R is the max total number of packets to be transmitted. (Note that this particular distribution comes from the arrival pattern as shown in Figure 2, and it is possible to design other patterns to adjust the relative distance of these distributions). Since M is just a sum of random variables for which we know the mean and variance, we then approximate the distribution of m with normal distribution:

$$m \sim \begin{cases} inactive: N(R, R) \\ normal: N(\frac{R}{2}(d_1 + 2), \frac{R}{2}(2d_1 + 3)) \\ event: N(R(d_1 + 1), R(2d_1 + 2)) \end{cases}$$

Once we have the distribution of detector m, we can then select thresholds t_n and t_e to satisfy desired bounds on notification false positive rate R_n and an event false positive rate R_c using the quantile function of normal distribution:

$$t_n = R + \sqrt{2R}\ erf^{-1}(1 - 2R_n)$$
$$t_e = \frac{R}{2}(d_1 + 2) + \sqrt{2R(2d_1 + 3)}\ erf^{-1}(1 - 2R_e)$$

After selecting thresholds according to the false positive rates, we can derive the false negative rate for classifying normal and event states. For simplicity, we take the max of these two as the failure rate for WIN:

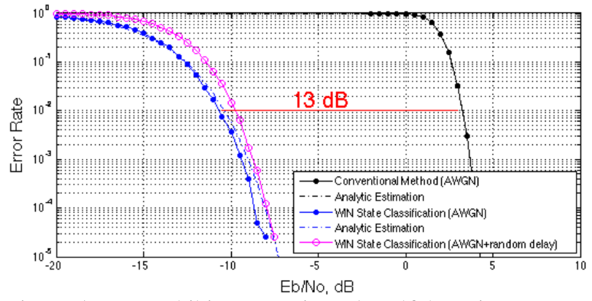


Figure 5. WIN exhibits approximately a 13dB gain over conventional approach under AWGN channel.

$$p(fail) = \frac{1}{2}\left\{1 + erf\left(\frac{t_e - R(d_1 + 1)}{\sqrt{2R(2d_1 + 2)}}\right)\right\}$$

We compare the WIN failure rate to that of the conventional approach derived earlier in Section 4 by evaluating failure rate at different *SNR*. As shown in Figure 5, WIN clearly outperforms the conventional approach.

6. Simulation

We present the simulation results on the error rate for the conventional system and the WIN proposal. The number of total packets (*R*) in a complete transmission is 20, and the number of bits per packet (*n*) is 80. Since CRC error becomes more likely when the packet size is larger, we select the smallest packet size for a wireless network to avoid bias against the conventional method. This size is 80 bits according to the specifications of Bluetooth LE [9]. We simulate with two channel models: AWGN channel and AWGN channel with uniform random packet delay.

The simulation results are shown in Figure 5. Under AWGN channel, WIN achieves error rates lower than 1% as long as the received SNR is greater than -10dB (see blue curve), while the conventional method has more than 1% error at 3 dB. In other words, there is roughly a 13 dB gain for WIN. Note that our analytic estimations match closely to the results obtained by simulation.

In the case where there are random packet delays due to multipath, WIN experience minor performance loss because of variations in the packet arrival pattern. In our simulation we assume a random delay up to 3 samples based on indoor environment, and the SNR loss is only about 0.5 dB (See magenta line in Figure 5). The conventional method process packets individually, and is therefore not affected by packet delays. Overall, WIN outperforms conventional method by a large margin.

7. Conclusion

Conventional network layering is provided to support modular design principles, but it is at the expense of losing information in each layer. For example, in the physical layer we loss information from demodulation and in the link layer we loss information when we toss the entire packet upon CRC errors. Furthermore, conventional design avoids utilizing prior knowledge because it is not always available. Such information loss and underutilization means a substantial drawback for applications that have stringent low-energy requirements. Via interference technology based on machine learning, WIN aims at making use of all information resulting from physical-layer matched filtering operations. In addition, WIN leverages designed-in separation between traffic patterns of different states of the sender, so the state classification can be tolerant to channel noise. For these reasons, we have shown that WIN can achieve 13 dB gains in terms of robustness against channel noise. Lowering the required signal strength at receiver by 13 dB translates to 4.5x range in free space. Our results may be useful for future ultra-low power designs for notification transmission over wireless channels.

ACKNOWLEDGEMENTS

This material is based on research sponsored in part by the Intel Corporation, and by the Air Force Research Laboratory under agreement number FA8750-10-2-0180. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the U.S. Government, or the Intel Corporation.

REFERENCES

[1] Capella, J.V.; Perles, A.; Bonastre, A.; Serrano, J.J. "Historical Building Monitoring Using an Energy-Efficient Scalable Wireless Sensor Network Architecture". Sensors 2011

[2] Smith P., Comparisons between Low Power Wireless Technologies, On www.csr.com/news/white-papers

[3] ZB Alliance, IEEE 802.15. 4, ZigBee standard, On www.zigbee.org, 2009

[4] Gomez C., Oller J., Paradells J., "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology", Sensors, 2012

[5] Liu Q.; Zhou S.; Giannakis, G.B., "Cross-Layer combining of adaptive Modulation and coding with truncated ARQ over wireless links," IEEE Transactions on Wireless Communications, 2004

[6] Lin, T-H and Kung, H. T. "Concurrent Channel Access and Estimation for Scalable Multiuser MIMO Networking. " 32nd

IEEE International Conference on Computer Communications (INFOCOM 2013) Mini-Conference, 2013.
[7] Yang J.; Yu K.; Gong Y.; Huang, T., "Linear spatial pyramid matching using sparse coding for image classification". CVPR 2009.
[8] Reed, I.S.; Gau, Y.L.; Truong, T. K, "CFAR detection and estimation for STAP radar," IEEE Transactions on Aerospace and Electronic Systems, vol.34, no.3, Jul 1998
[9] Bluetooth specification core version 4.0, On www.bluetooth.org/Technical/Specifications