Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems

Zhufan Wang and Guangyan Zhang, ***Tsinghua University;***
Yang Wang, ***The Ohio State University;*** **Qinglin Yang,** ***Tsinghua University;***
Jiaji Zhu, ***Alibaba Cloud***

https://www.usenix.org/conference/atc19/presentation/wang-zhufan

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems

Zhufan Wang[†], Guangyan Zhang[†,*], Yang Wang[‡], Qinglin Yang[†], Jiaji Zhu[§]
[†]*Tsinghua University,* [‡]*The Ohio State University,* [§]*Alibaba Cloud*

Abstract

This paper tries to accelerate data recovery in a large-scale storage system with minimal interference to foreground traffic. By investigating I/O and failure traces from a real-world large-scale storage system, we find that because of the scale of the system and the imbalanced and dynamic foreground traffic, no existing recovery protocols can generate a high-quality recovery strategy in a short time.

To address this problem, this paper proposes Dayu, a timeslot-based recovery protocol, which only schedules a subset of tasks which are expected to finish in one timeslot: this approach reduces the computation overhead and naturally can cope with the dynamic foreground traffic. In each timeslot, Dayu incorporates four key algorithms, which enhance existing solutions with heuristics motivated by our trace analysis.

Our evaluations in a 1,000-node real cluster and in a 25,000-node simulation both confirm that Dayu can outperform existing recovery protocols, achieving high speed and high quality.

1 Introduction

This paper describes our experience and methods to accelerate data recovery in Pangu [1] , a real-world large-scale storage system with 10K nodes and tens of TBs of storage per node.

As a cloud storage provider, AliCloud, the owner of Pangu, needs to make a promise of data durability to its customers (*i.e.*, the chance of data loss is smaller than a threshold). For marketing reasons, the owner has a strong motivation to improve data durability, so that its promise can be appealing compared to its competitors. This motivates us to investigate whether it is possible to accelerate data recovery in Pangu, because recovery speed is one of the determining factors of data durability [2].

Similar to previous works [3–7], Pangu divides data into chunks (usually tens of MBs), replicates these data chunks, and distributes these replicas to different nodes. When a node fails, Pangu re-replicates its data chunks: since the replicas of these chunks are distributed to different nodes, Pangu asks all these nodes to copy chunks in parallel [4, 8].

To re-replicate data chunks of the failed node, the recovery protocol needs to schedule a source, a destination, and a bandwidth for each of these data chunks. An ideal scheduling algorithm should achieve at least the following two goals: first, the algorithm should generate a high-quality strategy, which should allow data re-replication to be completed as soon as possible under the constraint that it has minimal impact on foreground traffic; second, the speed of the scheduling algorithm itself should be high enough so that it does not become the bottleneck of data recovery.

To understand the quality and speed of existing scheduling algorithms, we analyze the failure and I/O traces from a real deployment of Pangu. We find none of the existing algorithms can achieve both acceptable quality and acceptable speed, because of the following challenges:

- *Very-large scale*: the largest deployment of Pangu has more than 10K nodes and up to 72 TBs of storage (about 1.5M chunks) per node. Therefore, when a node fails, the algorithm needs to decide how to recover all these data chunks and each chunk has about 10K nodes as candidate destinations.
- *Tight time constraint*: given the scale of the system, data chunks of a failed node can be re-replicated with a high degree of parallelism. Our simulation shows that if the idle bandwidth can be fully utilized, the recovery can be finished within tens of seconds, which means the scheduling algorithm itself should complete within seconds.
- *Imbalanced foreground traffic and available data*: we find a two-fold imbalance, which poses challenges to the quality of scheduling. First, a number of nodes can have significantly heavier foreground traffic than the others; and second, some nodes can have more data chunks available for re-replication.
- *Dynamic foreground traffic*: the foreground traffic can change dramatically over time. To cope with such dynamic traffic, the recovery protocol needs to adjust its

[*]Corresponding author: gyzh@tsinghua.edu.cn

plan when it observes a significant change in the foreground traffic, which again calls for fast scheduling.

Our simulation of existing scheduling algorithms shows that, on the one hand, simple and decentralized algorithms like random selection or best-of-two-random [9] can finish scheduling quickly (*i.e.*, high speed), but they often cause a small number of nodes to be overloaded, increasing the recovery time and impairing the performance of foreground traffic (*i.e.*, low quality). On the other hand, sophisticated and centralized algorithms, such as Mixed-Integer Linear Programming [10–12], can effectively utilize available bandwidth and avoid overloading a node (*i.e.*, high quality), but they can take prohibitively long to compute a plan given the scale of our target system (*i.e.*, low speed).

This paper proposes Dayu, a high-speed and high-quality recovery protocol for large-scale, imbalanced, and dynamic storage systems. The key idea of Dayu is motivated by the observation that, to cope with dynamic foreground traffic, we need to periodically monitor the foreground traffic and adjust the recovery plan: in such a design, scheduling for all data chunks of the failed node together is both computationally heavy and unnecessary, since the plan is likely to be adjusted later. Following this observation, Dayu incorporates a timeslot-based solution: it divides time into multiple slots, whose length is determined by how frequent the underlying storage system monitors and reports idle bandwidth; based on such report, Dayu tries to schedule a subset of chunks so that they can be re-replicated within the current timeslot; if the actual re-replication of some chunks takes longer than expected for whatever reason, Dayu will re-schedule them in the next timeslot.

This approach brings two benefits: first, it reduces the computation overhead of scheduling because in each timeslot, the algorithm only needs to schedule a subset of tasks (about one third on average in our experiments). Second, this solution can naturally cope with the dynamic foreground traffic because Dayu's decision is based on the information collected at the beginning of each timeslot.

To realize this idea, Dayu incorporates four key techniques, which enhance existing algorithms based on our observations:

- *Greedy algorithm with bucket convex-hull optimization to schedule tasks*: Dayu uses a greedy algorithm to iteratively choose the most under-utilized candidate as the source and destination for each task, till it finds enough tasks to fill a timeslot. To reduce the computation overhead, Dayu incorporates the convex-hull optimization [13] and further proposes a bucket approximation to reduce the size of the candidate set.
- *Prioritizing nodes with high idle bandwidth but few available chunks*: Our observation shows that such nodes are likely to get under-utilized, if the scheduling algorithm decides to replicate their chunks from other nodes. Therefore, Dayu enhances the aforementioned greedy algorithm with the following heuristic: if a chunk to be re-replicated has a replica in such a prioritized node, Dayu will assign the node as the source.
- *Iterative WSS to allocate bandwidth for each task*: To minimize the completion time of chosen tasks, Dayu enhances the weighted shuffle scheduling algorithm (WSS) [14]: in each iteration, Dayu uses WSS to identify the bottlenecks in the remaining tasks, assigns a weighted fair share of bandwidth to each task correspondingly, and removes the bottleneck tasks and allocated bandwidth.
- *Re-scheduling stragglers*: Straggler tasks will inevitably occur due to mis-prediction of the foreground traffic or unexpected hardware faults, so Dayu has to re-schedule them in the next timeslot. Straggler tasks are different from new tasks, since we prefer keeping their destinations unchanged: otherwise, we will lose their existing progress. Dayu first estimates whether it is worth changing their destinations, and then re-computes their sources and allocated bandwidth.

Our evaluation of Dayu on a real deployment of 1,000 nodes shows that, compared to Pangu, Dayu increases the recovery speed by 2.96× and increases the p90 latency (i.e., tail latency at 90^{th} percentile) of the foreground traffic during recovery by only 3.7%. Our simulation shows that Dayu outperforms various existing solutions and can scale to a cluster of 25K nodes.

2 Background and Observations

2.1 Background of Pangu

Pangu is the underlying storage system of AliCloud, one of the largest public cloud providers in Asia [1]. Pangu inherits the classic distributed file system architecture from previous works like GFS [3], HDFS [5], Cosmos [6], and Azure [7]. It splits data into multiple chunks (the most common chunk size is 64MB) and stores data chunks on a large number of data servers called *ChunkServers*. A metadata server called *MetaServer* maintains the metadata of the distributed file system, such as the locations of data chunks. Given its very large scale, Pangu incorporates multiple *MetaServers*, each responsible for a subset of metadata [15–17]. Besides, Pangu incorporates a *RootServer* to route clients to the corresponding *MetaServer*. To achieve uniform data distribution, Pangu uses random or weighted random mechanism to place data on different *ChunkServers*.

Like most existing systems, Pangu replicates data chunks (most chunks have three replicas) so that if a node fails, Pangu can recover its data chunks by copying from other replicas. For each data chunk to be recovered, Pangu needs to choose a source and a destination for data copy: there are usually a few candidate sources depending on the number of replicas and a

large number of candidate destinations. The current version of Pangu randomly picks a source and a destination for each data chunk to be recovered.

In the current deployment of Pangu, we observe that the network bandwidth is usually the bottleneck when performing such data recovery: most Pangu nodes are equipped with 1Gb or 10Gb Ethernet, whose bandwidth is smaller than the aggregate disk bandwidth; the deployment of high-speed devices, such as Infiniband, is limited due to cost reasons. Pangu's core network switches are organized using CLOS topology or fat-tree topology [18–21], so that there is no oversubscription. The core-to-rack link may be oversubscribed depending on the configuration: if the link is oversubscribed, the rack switch is usually the bottleneck; if not, the NICs of end-hosts are the bottlenecks.

During data recovery, Pangu is still servicing foreground applications, which may contend for network bandwidth. To limit the interference of data recovery on foreground traffic, Pangu provides a mechanism to limit the bandwidth utilization of one or a group of links on a node. With this mechanism, we can set a limit on the bandwidth of the recovery traffic, depending on how much interference one is willing to tolerate and the bandwidth of the foreground traffic. In Dayu, we limit the bandwidth of the recovery traffic on each node to be

$$B_{recover} = max(\alpha \times B_{total} - B_{foreground}, B_{min}) \quad (1)$$

In this equation, B_{total} is the total bandwidth of the node; $B_{foreground}$ is the bandwidth of the foreground traffic; and α is a parameter to control the interference of the recovery traffic on the foreground traffic. We set α to be 75% and our experiments show that using this setting will incur negligible impact on p90 latency of the foreground traffic. As a storage system mainly designed for large files, Pangu does not aim at optimizing extreme tail latency (e.g. 99.9 percentile [22]), so this setting can satisfy our requirement, and if one is targeting even smaller interference, he/she can further decrease α. B_{min} is the minimal bandwidth the node will assign for recovery, which is to ensure that recovery will not be too slow. Both Pangu and Dayu set B_{min} to 30MB/s.

2.2 Observations

In this subsection, we analyze the workload and data placement from one deployment of Pangu to understand how they affect data recovery. We acquire such information from a data center of approximately 3500 nodes, each with two 1G NICs and 11 2TB hard drives. In this case, the aggregate bandwidth of hard drives is larger than that of the NICs. The storage system mainly serves online data processing service (*ODPS*), including MapReduce and data query. What we analyze includes 1) a checkpoint of a *MetaServer* in April 2018, which records the metadata related to the size and distribution of the data chunks, and 2) the trace of the foreground traffic and background recovery traffic in the coming week. Unless otherwise noted, our simulation experiments are on this 3500-node cluster throughout this paper. We study the scalability of Dayu beyond 3500 nodes in Section 5.2.

We make the following observations from the analysis:

Observation 1 *Each node stores hundreds of thousands of chunks.*

Figure 1(a) shows the CDF of the number of chunks on each node. We can observe that a majority of the nodes have around 250K chunks per node. This observation suggests two things: first, a recovery protocol needs to schedule how to recover so many chunks when a node fails. Second, when one node fails, each of the remaining nodes will participate in the re-replication of about 70 chunks on average (250K/3500).

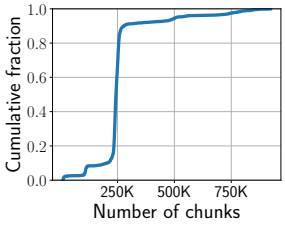
Observation 2 *The foreground traffic consumes less than half of the bandwidth on average. If all available bandwidth (computed using Equation 1) can be used for recovery, the system can recover 250K chunks in 51 seconds on average.*

We calculate the optimal recovery time for 50 different cases, assuming all available bandwidth can be utilized, and present the CDF of the recovery time in Figure 1(b).

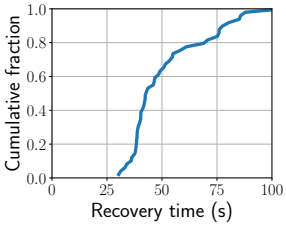
This observation suggests that, although there are a large number of chunks to recover for each node failure, the highly parallel recovery in a large-scale system can recover these chunks in a short time, which calls for fast scheduling during the recovery protocol. However, the actual recovery in the trace often takes 2-4 minutes, i.e. $2.35-4.70\times$ of the ideal recovery time, which motivates our further investigation.

Observation 3 *The foreground traffic is experiencing significant short-term load imbalance.*

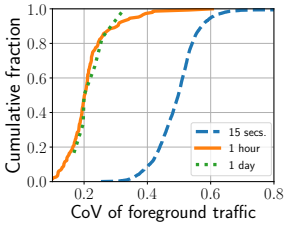
The trace we analyze records the foreground bandwidth of each node every 15 seconds. To understand whether the foreground traffic is balanced, we compute the coefficient of variation (CoV, standard deviation as a percentage of the mean) of foreground bandwidth in each timeslot, which is a standard metric to measure the variation of values. Then we draw the distribution of CoVs of different timeslots in Figure 1(c) and Figure 1(d). As shown in this figure, the CoVs of most timeslots are between 0.4 and 0.6, which is quite significant. Interestingly, if we measure such imbalance in a coarser granularity (i.e. one hour and one day), the imbalance becomes much smaller. Such results indicate that the system is relatively load balanced in a long term, but more imbalanced in a short term, which creates a challenge for data recovery: traditional load balancing techniques, such as data migration, mainly targets long-term imbalance, because they cannot run very frequently; data recovery, however, is mainly affected by short-term imbalance, because it can finish within tens to a few hundred seconds. This observation suggests that our recovery protocol must take such short-term imbalance into consideration, without relying on load balancing techniques.



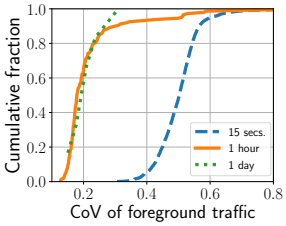
(a) OB1: CDF of the number of chunks on each node



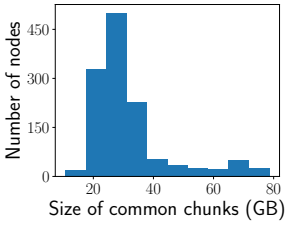
(b) OB2: CDF of the optimal recovery time with 50 cases



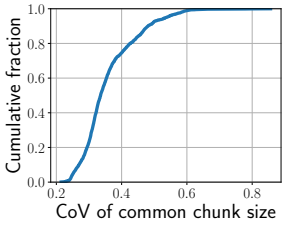
(c) OB3: CoV of outgoing bandwidth utilization



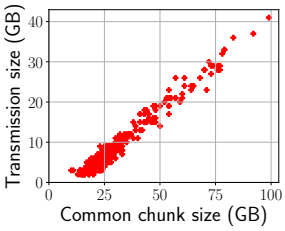
(d) OB3: COV of incoming bandwidth utilization



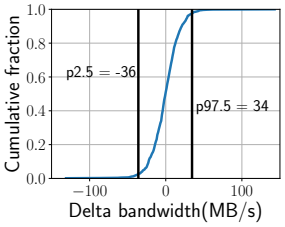
(e) OB4: Distribution of SC_j^{100}.



(f) OB4: CoV of SC_j^i for each node i



(g) OB4: Correlation between recovery traffic of node j and SC_j^{100}



(h) OB5: Bandwidth utilization variance.

Figure 1: Observations from a 3500-node real-world system

Observation 4 *Replicas of chunks on a given nodes are distributed unevenly among the other nodes.*

To understand how replicas of chunks on a given node are distributed among the other nodes, we define SC_j^i as the size of the chunks held by both node i and node j, which shows how much data node j can provide as source during recovery if node i fails.

We first sample a specific node $i = 100$. Figure 1(e) shows the distribution of SC_j^{100} for different j values. We can see that the histogram of the distribution fits the bell curve: this is actually mathematically provable (i.e. Central Limit Theorems) if we assume chunk placement is random. To understand such imbalance in the whole cluster, for each node i, we calculate the CoV of all the SC_j^i values and then we draw the CDF of CoVs of all nodes in Figure 1(f). One can observe that for a large portion of nodes, the distribution of SC_j^i is not balanced.

To understand how such imbalance affects recovery, we simulate the failure of node 100 with Pangu's random node selection strategy (Figure 1(g)) and find that there is a strong correlation between the size of outgoing recovery traffic of node j and SC_j^{100}. That means a node with a few (many) common chunks with the failed node will do little (much) work during recovery, but if the node has much (little) available bandwidth, it will get under-utilized (overloaded).

Observation 5 *Foreground traffic usually fluctuates within 14.4% of max bandwidth, but sometimes can change dramatically.*

Figure 1(h) shows how one node's foreground traffic changes in 5 hours. The delta bandwidth is the difference of the average bandwidth utilization in two adjacent timeslots. We can observe that in more than 95% of the cases (between "p2.5" and "p97.5" in Figure 1(h)), the absolute delta bandwidth is lower than 36MB/s, which is 14.4% of the maximum bandwidth (250 MB/s since each node has two 1Gb NICs). However, in the remaining 5% cases, the delta bandwidth can reach up to two thirds of the maximum bandwidth. Although the percentage of such extreme cases is small, they frequently happen in recovery, because the highly parallel recovery usually involves many nodes. Our simulation shows that they can create stragglers in recovery and thus are one of the major reasons why recovery speed is not ideal.

3 Dayu Overview

We call the re-replication of one data chunk a "recovery task" in the rest of the paper. Dayu achieves fast data recovery and low application interference by introducing a centralized scheduler called an *ObServer*, which performs timeslot-based recovery task scheduling.

Dayu assumes all the data servers periodically report their chunk placement and network utilization to the *ObServer*, and all the metadata servers send information of the recovery tasks to the *ObServer*. The rest of this paper presents the *ObServer*'s scheduling algorithm, which decides the source, the destination, and the bandwidth of a recovery task.

To achieve high-speed and high-quality scheduling, the key idea of Dayu is to schedule recovery tasks in multiple batches, instead of scheduling all of them together. This design choice is motivated by several reasons: first, since each node is usually involved in tens of recovery tasks (Observa-

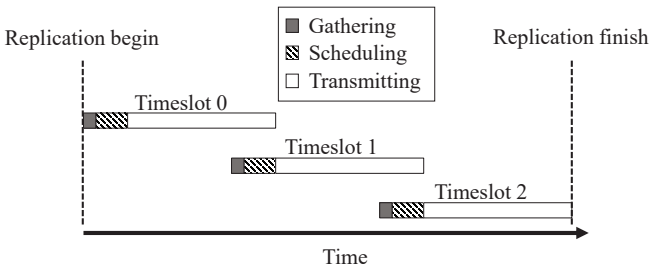


Figure 2: Timeslot-based scheduling in Dayu

$T_{timeslot}$	Length of a timeslot
$B^i_{recover\_in/out}$	Node i's available incoming/outgoing bandwidth for recovery (by Equation 1)
s_t	Size of recovery task t
$c^i_{in/out}$	Total size of incoming/outgoing recovery tasks assigned to node i
α, β	Parameters

Table 1: Denotations

tion 1), scheduling tasks in multiple batches can still allow each node to fully participate in each batch and thus fully utilize its available bandwidth; second, scheduling tasks in batches can naturally cope with dynamic foreground traffic and infrequent measurement errors, because when observing any changes in the foreground traffic, Dayu can make adjustment in the next batch; finally, scheduling tasks in batches naturally reduces the computation overhead of the scheduling algorithm, because for each batch, the algorithm only needs to schedule a subset of tasks.

To implement this idea, as shown in Figure 2, Dayu divides the whole recovery time into multiple fixed-length time slices (called *timeslots* throughout the paper). At the beginning of a timeslot, the *ObServer* collects the latest state of the data servers. Using the state obtained, the *ObServer* chooses and schedules a subset of recovery tasks in this timeslot, including those recovery tasks scheduled in the last timeslot but unfinished yet. To fully utilize the available bandwidth, Dayu overlaps multiple timeslots so that the information gathering and task scheduling of slot n is executed before the end of slot $n-1$. The length of a timeslot is determined by how frequently the underlying storage system collects and reports state.

As mentioned in Section 2.1, the bottleneck of data recovery is either the NICs of the end hosts or the rack switch, and to simplify description, the following text assumes the NICs of the end hosts are bottlenecks, and one can easily extend it to support bottleneck rack switches. Table 1 lists the denotations used in the paper.

Goals. Dayu tries to achieve the following goals.

- **Goal 1: Utilize the available bandwidth as much as possible.** This is a natural goal to minimize the overall recovery time. If we were to fully utilize the available bandwidth in one timeslot, the total size (S) of the chunks that can be replicated in the timeslot would be:

$$S = min(\sum_{i \in Nodes} B^i_{recover\_in}, \sum_{i \in Nodes} B^i_{recover\_out}) \times T_{timeslot} \quad (2)$$

- **Goal 2: Finish as many tasks as possible in the target timeslot.** We hope that the scheduled tasks can actually finish within the target timeslot: otherwise, we have to re-schedule them again, which increases the computation overhead. This goal may look similar to the first one, but it is not: the first goal suggests us to oversubscribe the network bandwidth (i.e. schedule more tasks than the bandwidth can handle), so that if the foreground traffic drops, we can still utilize such extra available bandwidth; the second goal, however, suggests us to undersubscribe the network bandwidth so that if the foreground traffic increases, we can still finish the scheduled tasks. Therefore, Dayu has to make a trade-off between these two goals.
- **Goal 3: Minimize the chance of significant stragglers.** Because we cannot accurately predict the future foreground traffic, stragglers will inevitably occur. We prefer many small stragglers to a few significant stragglers, because many small stragglers can be re-scheduled and executed in parallel to minimize the recovery time. However, this goal obviously contradicts with the second goal, so Dayu has to make a trade-off as well.

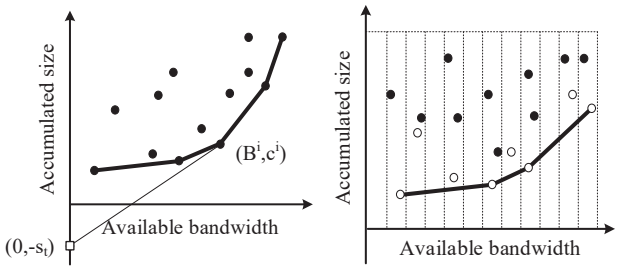
Overview of Dayu's algorithm. To achieve these goals, Dayu incorporates four key techniques, by enhancing existing algorithms with heuristics and approximations motivated by our observations: 1) a greedy algorithm with bucket convex hull optimization to select the source and the destination for each recovery task (§4.1); 2) a heuristic-based algorithm to prioritize nodes with a few common chunks with the failed node but a high available bandwidth (§4.2); 3) an iterative WSS algorithm to assign bandwidth for each task (§4.3); and 4) a heuristic-based algorithm to minimize the cost of re-scheduling straggler tasks (§4.4).

4 Design of Dayu

4.1 Selecting Source and Destination

Dayu iteratively scans all tasks and determines the source and the destination for each task, till it can find enough tasks to fill S (Equation 2). The candidate sources of a task include all nodes which hold a replica of the corresponding chunk; the candidate destinations of a task include all nodes which are not in the same rack of its sources.

To achieve the goals given in Section 3, Dayu incorporates a greedy algorithm: for each task, Dayu chooses the most under-utilized node in its candidate sources and destinations;



(a) Convex hull optimization (b) Bucket approximation

Figure 3: Reduce computation overhead with dynamic convex hull optimization. B^i and c^i are short for $B^i_{recover\_in}$ and c^i_{in}.

if Dayu finds that even the most under-utilized candidate is going to be saturated, Dayu will skip this task.

The first question we need to answer is how to quantitatively measure the utilization of a node. We have tried several options, and through simulation, we decide to use the expected task finish time $\frac{c_{in/out}}{B_{recover\_in/out}}$ ($c_{in/out}$ is the total size of the incoming/outgoing tasks assigned to this node) as the metric to evaluate the utilization of a node, because this metric achieves a nice balance between our first two goals.

Therefore, when choosing the source for task t, Dayu scans all its candidate nodes and chooses the one with the minimal $\frac{s_t+c_{out}}{B_{recover\_out}}$ as the source. Such scanning is not computationally expensive, since most chunks have three replicas and one of them is already lost. Afterward, Dayu checks whether assigning task t to the source will saturate the source, i.e. its $\frac{s_t+c_{out}}{B_{recover\_out}} > T_{timeslot}$: if so, Dayu will drop task t, because this means there is no way to complete task t in this timeslot.

Likewise, when choosing the destination for task t, Dayu chooses the node with the minimal $\frac{s_t+c_{in}}{B_{recover\_in}}$. However, naively scanning all candidate destinations is computationally heavy, since the number of candidate destinations is large. To make things worse, greedy algorithms cannot be parallelized because each iteration depends on the result of the previous iteration. Our simulation on a 3500-node cluster shows that naively scanning all candidate destinations for each task can only achieve a speed of less than 30,000 tasks per second. Since our statistics shows that in one timeslot, Dayu can usually complete 60,000—150,000 tasks, this means naively scanning itself will take 2-5 seconds, which is not ideal. To address this challenge, we incorporate the dynamic convex hull optimization to accelerate this computation.

One can refer to [13] for the formal description of the convex hull optimization, and here we present an intuitive description. For each surviving node i, we draw a point $(B^i_{recover\_in}, c^i_{in})$ in Cartesian coordinate system, as shown in Figure 3(a). Then for task t with size s_t, we draw another point $(0,-s_t)$ in Figure 3(a). Afterward, we draw a line from $(0,-s_t)$ to each other point: since the slope of each line is $\frac{c^i_{in}+s_t}{B^i_{recover\_in}}$, finding the destination node for task t is equivalent to finding the line with the lowest slope.

We can maintain a dynamic convex hull to quickly search the line with the lowest slope. In a two-dimensional space, a convex hull is like a rubber band that wraps all the points tightly, where the lower convex shell is the lower part of this convex hull. We refer the point set of the lower convex shell as H (here we connect points in H together to form the lower convex shell as shown in Figure 3(a)). The points in H are connected counterclockwise. Then for a point p_h with precursor point and successor point in set H, the slope of line $p_{h-1} \rightarrow p_h$ must be less than or equal to the slope of line $p_h \rightarrow p_{h+1}$. We can find the node c in set H whose connection with point $(0,-s_t)$ has the smallest slope using binary search and the time complexity is $O(\log|H|)$.

After Dayu assigns task t to node i, its c^i_{in} is incremented by s_t. Therefore, Dayu needs to adjust the point of node i as well as the lower convex shell H: when a point p_h in H moves up, Dayu identifies the precursor (p_{h-1}) and successor (p_{h+1}) of p_h in original H, and scans all the points between them to find the new member(s) of H. The convex hull optimization reduces the complexity of scanning destination nodes from linear to sub-linear, without affecting the results of the greedy algorithm.

We further propose an approximate solution to reduce the candidate set of the lower convex shell, in turn boosting the speed of the algorithm. As shown in Figure 3(b), we divide the range of available incoming bandwidth into multiple equal-sized buckets. If nodes i and j are in the same bucket, they are considered to have approximately identical available bandwidth, i.e. $B^i_{recover\_in} \approx B^j_{recover\_in}$. Without loss of generality, we suppose $c_i > c_j$. Then node i cannot be the member of the lower convex shell. Therefore, only the lowest node within the same bucket can become the member of the lower convex shell. All those lowest nodes (hollow circles in Figure 3(b)) form a reduced candidate set, denoted C. We can construct the convex shell H from this reduced candidate set C, instead of the full set of nodes. After Dayu assigns a task to a node, it adjusts the point of this node as well as the reduced candidate set.

The bucket size determines the reduction degree of the bucket approximation. We use 1 MB/s as the bucket size in our experiments, and our simulation shows an average reduction factor of 22.8, and as a result, Dayu can complete selecting sources and destinations for about 210,000 chunks within one second—this is seven times faster than naive scanning.

Such bucket approximation certainly brings inaccuracy to the greedy algorithm, but such inaccuracy already exists as a result of measurement errors and fluctuation of foreground traffic. Therefore, as long as the bucket size is small, our approximation should not significantly increase such inaccuracy.

4.2 Prioritizing Underemployed Nodes

Our simulation on our greedy algorithm reveals the same problem as our Observations 3 and 4: nodes with high available bandwidth but only a small number of available chunks are likely to get under-utilized, which violates our first goal. We call them *underemployed nodes* in the rest of the paper. For example, suppose node A has an available outgoing bandwidth of 50MB/s and can be the source of Tasks 1 and 2; node B has an available outgoing bandwidth of 60MB/s and can be the source of Tasks 1-4; all tasks have the same size. In this example, the optimal schedule should let A be the source of Tasks 1 and 2, and B be the source of Tasks 3 and 4. However, if our greedy algorithm scans Task 1 first, it will assign it to node B, because B has more available bandwidth than A at this moment.

This observation suggests that, for a chunk which has a replica in an underemployed node, it's better to use the underemployed node as the source. To achieve this goal, we incorporate a distribution-driven prioritizing strategy: the *ObServer* first sorts all the nodes according to their available outgoing bandwidth in descending order, and sorts all the nodes according to their total sizes of common chunks in ascending order. Then, the *ObServer* picks the first β (5% in our typical settings) nodes from those two node lists respectively to form two sets, and gets the underemployed node set by computing the intersection of those two sets. Next, the *ObServer* selects all the recovery tasks that have replicas in the underemployed nodes, and puts them in a queue called "prioritized queue"; the *ObServer* puts the rest of the tasks in another queue called "normal queue".

We modify our greedy algorithm (§4.1) to incorporate this heuristic: the *ObServer* will first scan tasks in the prioritized queue and directly use the corresponding underemployed server as the source, instead of using the most under-utilized candidate. There are two corner cases: 1) it is possible that a prioritized task has replicas in more than one underemployed servers. In this case, the *ObServer* chooses the most under-utilized one among them; 2) though rare, it is possible that the underemployed server is saturated. In this case, the *ObServer* degrades the prioritized task into the normal queue, so that later we can still try its non-prioritized candidates.

Overhead. When searching underemployed nodes, Dayu maintains two heaps, whose keys are the available outgoing bandwidth and the total size of common chunks respectively, and whose values are the IDs of nodes. The *ObServer* will first build these two heaps, which is an $O(n)$ operation (n is the number of nodes) [23], and then pop 5% entries from these two heaps, with each pop an $O(\log n)$ operation. Our experiment shows that heapifying 10,000 entries and then popping up 5% of them only take a few miliseconds.

4.3 Allocating Bandwidth for Each Task

Given the source and destination of each recovery task, we need to answer how fast each task should proceed. A naive solution is to set a coarse-grained limit on all tasks within one node using $B_{recover\_in/out}$ and let them compete for bandwidth. However, our experiments have revealed two problems with this approach: first, this approach may cause a congestion when the source's outgoing limit is larger than the destination's incoming limit. Although TCP can resolve such congestion eventually, it will cause packet drops and slow down recovery. Second, the competition may cause one task to be significantly slower than others, causing a significant straggler and violating our third goal.

Therefore, in this step, Dayu tries to set a constant rate for each task in one timeslot, with the goal of maximizing bandwidth utilization. Recall that we assume the NICs of the end hosts are the bottlenecks, so this step only considers the bandwidth utilization at the end hosts. Even so, this is still a challenging problem, since allocating bandwidth for a task will consume the bandwidth on both sides.

Dayu's solution is based on weighted shuffle scheduling (WSS) [14], a mature network scheduling algorithm designed for scheduling large data flows like data shuffle in MapReduce [24]. The key idea of WSS is that, to finish all the pairwise transfers at the same time, it guarantees that 1) transfer rates are proportional to data sizes for each transfer, and 2) at least one link is fully utilized. With WSS, only the bottleneck links (quite a minority) are fully used, while all the others have an amount of bandwidth left. In our scenario, however, WSS is not ideal: when considering unpredictable growth in foreground traffic, which may cause a non-bottleneck link to become a bottleneck in the middle of a timeslot, WSS may cause a waste of bandwidth, because Dayu could utilize more bandwidth of this link at the beginning.

To this end, Dayu introduces an iterative WSS solution to allocate bandwidth for each task. Its key idea is that, without delaying the bottleneck tasks, we should finish other tasks as early as possible, so as to reduce their completion time and to improve bandwidth utilization. Following this idea, if there is any remaining bandwidth after running one iteration of WSS, Dayu will use another iteration of WSS to identify the next bottleneck and allocate the remaining bandwidth.

To be specific, Dayu maintains a remaining incoming and outgoing bandwidth $B^i_{remain\_in}$ and $B^i_{remain\_out}$ for each node, whose initial values are $B^i_{recover\_in}$ and $B^i_{recover\_out}$. In each iteration, Dayu first finds the node with the longest $\frac{c^i_{in}}{B^i_{remain\_in}}$ or $\frac{c^i_{out}}{B^i_{remain\_out}}$, denoted T^*: the corresponding tasks are the bottlenecks. Then, Dayu allocates $B_t = \frac{s_t}{T^*}$ bandwidth to each task t, indicating that to minimize the completion time, the bottleneck tasks must be assigned a weighted fair share of the bandwidth, such that the weight of the share is proportional to s_t. Afterward, Dayu updates the remaining bandwidth as

$B^i_{remain\_in} -= \frac{c^i_{in}}{T^*}$ and $B^i_{remain\_out} -= \frac{c^i_{out}}{T^*}$ for each node i, removes the bottleneck tasks from their corresponding nodes, and updates the $c_{in/out}$ values of these nodes. Then Dayu moves to the next iteration with the remaining tasks, till there are no tasks remaining or the remaining tasks have an acceptable transmission time (i.e., less than or equal to the length of a timeslot) with their allocated bandwidth. Note that if a task goes through multiple iterations, its allocated bandwidth is the sum of the allocated bandwidth in each iteration.

Iterative WSS overcomes the drawbacks of WSS: since iterative WSS tries to allocate all bandwidth, it is not possible for the system to waste bandwidth when there are tasks that can utilize such bandwidth.

Our experiment shows that for a 3500-node cluster, each iteration will take at most 15 *ms*. Since dynamic convex hull node selection algorithm keeps the "$\frac{c}{B}$" values of most nodes to be close, the iterative WSS algorithm can usually finish within five iterations (i.e. 75 *ms*), which is acceptable.

4.4 Re-scheduling Straggling Tasks

Due to inaccurate workload estimation, sub-optimal scheduling, hardware exceptions, and etc., some tasks could not be finished at the end of one timeslot. Dayu has to re-schedule such straggler tasks in the next timeslot, but cannot simply treat them as new tasks, because changing the destination of one straggler task requires re-transmitting the task from the beginning, causing waste of bandwidth. Therefore, Dayu should avoid changing the destination when possible—this is a constraint new tasks do not have.

Identifying stragglers. Recall that Dayu overlaps different timeslots so that the scheduling phase of the current timeslot happens a short period of time (denoted as $T_{schedule}$) before the end of the last timeslot (Figure 2). Therefore, Dayu has to predict which tasks will become stragglers: for one unfinished task, Dayu uses its speed so far to estimate its speed till the end of the last timeslot; if the task cannot finish given the estimated speed, Dayu puts those tasks into a straggler set.

Prediction can certainly be inaccurate. If Dayu marks a task as a straggler but it actually finishes with the last timeslot, the corresponding nodes will simply ignore the new transmission plan scheduled by Dayu. Conversely, if a task is not marked as a straggler but it cannot finish within the last timeslot, the corresponding node will not get a new transmission plan, and thus will stick with the old plan. Both cases may cause inefficiency, but since $T_{schedule}$ is much smaller than $T_{timeslot}$, these two kinds of misidentification have little impact in our experiments.

Scheduling stragglers. First, Dayu will check whether the straggler set itself will saturate some nodes in the current timeslot. If any, the *ObServer* iteratively evicts the least finished task from each saturated node until it is no longer saturated. Those evicted tasks are categorized into two groups:

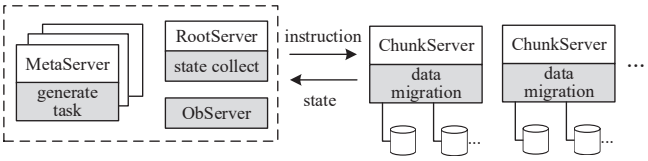


Figure 4: Our implementation of Dayu on Pangu. Gray boxes stand for functions or components Dayu adds to Pangu.

tasks evicted from their sources and tasks evicted from their destinations. They are rescheduled in different manners.

- For each straggler task in the first group, the *ObServer* chooses a source and a transfer rate (the same as a new task), while keeping the destination unchanged, which means the task can resume from its current progress.
- For each straggler task in the second group, the *ObServer* reschedules it as a new task.

For unevicted straggler tasks, Dayu keeps their sources and destinations unchanged, and allocates the bandwidth with the iterative WSS algorithm. They can resume from their current progress.

Compared to treating stragglers as new tasks, Dayu tries to minimize re-transmitting data, since it only changes the destination of the second group of stragglers (quite a minority) and re-transmits their data. Compared to letting stragglers continue with their original plans, our experiments show that the introduction of straggler adjustment improves the overall recovery speed by 15.6%.

It should also be noted that how to detect and report slow hardware is an orthogonal problem [25–27]. Dayu assumes the system has some mechanism to measure and report the actual bandwidth of each node.

5 Evaluation

Our evaluation tries to answer the following questions:

1. How fast can Dayu complete one typical full node recovery and how much interference does Dayu introduce between background and foreground? (§5.1)
2. Could Dayu scale to even larger systems? (§5.2)
3. In Dayu, how much benefit does each key technology bring? (§5.3)
4. How does the setting of the parameters affect the performance of Dayu? (§5.4)

Implemention. We implement Dayu upon Pangu, by modifying *MetaServers*, *RootServer*, and *ChunkServers* of Pangu and introducing Dayu's *ObServer* into Pangu, as shown in Figure 4. The *ObServer* is aware of the information of all the recovery tasks as well as the global information provided by the *RootServer*, such as the chunk placement and the network utilization at each *ChunkServer*. As Pangu monitors and reports the states of *ChunkServers* every 15 seconds, the

timeslot length of this implementation is set to 15 seconds. Upon detection of a node failure in Pangu, *MetaServers* report all the data chunks of the failed node to the *ObServer*, which schedules how to re-replicate these data chunks. Afterwards, the *ObServer* instructs the *ChunkServers* to execute recovery tasks (i.e. re-replicate data chunks). Finally, after a recovery task is completed, the *ObServer* updates the *MetaServers* to reflect the locations of the newly-re-replicated chunk.

Testbed. We have deployed the Pangu-based Dayu implementation on a 1000-node cluster. Each node has two 12-core Intel E5-2630 processors, 96GB DDR4 memory, two 10Gbps NICs, 10 or 11 2TB hard disks, and Linux 3.10.0. Since our traces are collected from a cluster with 1Gbps NICs but our testbed is equipped with 10Gbps NICs, we add a traffic control to our testbed so that each NIC can only use 1Gbps bandwidth.

We have also built a simulation environment to test Dayu with the scale of 3,500 nodes or more. We run the simulation in a server with two 16-core Intel E5-2620 processors, 64GB DDR4 memory, and Linux 3.10.0.

Methodology. For experiments on real-world systems, we trigger data recovery by shutting down one *ChunkServer*. When performing recovery, we replay the trace collected from the real-world cluster system (§2.2). Since our testing cluster is smaller than the cluster where the trace is from, we reshape the trace to fit the cluster size by trimming or redirecting some requests, while keeping the ratio of read and write, the pressure on each node, and the degree of imbalance among nodes [28]. We record both the recovery time and the interference between the foreground and recovery traffic, which is measured by comparing the p90 latency (i.e., tail latency at 90^{th} percentile) of the foreground requests with and without recovery traffic.

In the simulation experiments, we simulate the failure of a *ChunkServer* by sending its chunk information to Dayu. Since we do not actually run the system, we need to simulate the interference between the foreground and the recovery traffic. Due to the scale of the system, request level simulation takes very long, so we use flow level simulation as in [29, 30]. It simulates the bandwidth utilization of each link and periodically updates the utilization according to the foreground and recovery traffic information. We define the interference factor as the ratio between the overload traffic size and the link bandwidth, as follows:

$$B^i_{overload} = max(B^i_{recover} + B^i_{foreground} - 75\% \times B^i_{total}, 0) \quad (3)$$

$$F_{interference} = \frac{\sum_{i \in Nodes} B^i_{overload}}{\sum_{i \in Nodes} B^i_{total}} \quad (4)$$

The reason we define such an interference factor is that if the total bandwidth utilization exceeds 75% of the NIC's bandwidth, the foreground latency will increase significantly. To quantitatively understand this simulated interference factor, we map them to the p90 latency in the real-world experiments (§5.4): the short conclusion is that an interference factor smaller than 2% indicates very small interference and a factor close to or larger than 6.5% indicates very large interference. In our simulation experiments, we simulate 50 failure cases by randomly choosing 50 pairs of failed nodes and their failure time. For each algorithm, we simulate its performance on all the 50 cases and report its average performance numbers.

In our following experiments, Figure 5 presents the results from the real-world systems and the other figures present the results from the simulation experiments.

Comparison. In the experiments on the real-world systems, we compare Dayu with Pangu's original re-replication strategy, which adopts disk utilization aware random data placement and static rate control. We use three configurations Pangu-slow (limit recovery traffic to 30MB/s, which is the default configuration in production systems), Pangu-mid (90MB/s), Pangu-fast (150MB/s) as the baselines.

In the simulation experiments, we compare Dayu with different scheduling algorithms used in state-of-the-art systems (Table 2), with the exception of MCMF since its optimized solver is not open sourced. For fairness, we keep the node prioritizing and straggler adjustment part of Dayu, and plug in different node selection and bandwidth allocation algorithms. Specifically, when selecting the destination of recovery tasks, we compare Dayu's bucket dynamic convex hull algorithm (C) to the following algorithms: 1) *Random* (R), which randomly selects a node as the transmission source and destination; 2) *Best-of-two-random* (B2R), which first chooses two *ChunkServers* randomly, and then picks the lighter-loaded one as the source or destination [4, 9]; 3) *Weighted random* (WR), which uses the available bandwidth as the weight to randomly select a node; 4) *Greedy1* (G1), which scans all candidate *ChunkServers* as [31, 32], then in our scenario finds the one with minimal $\frac{c}{B}$. 5) *Greedy2* (G2), which chooses the lightest-loaded *ChunkServer*, by maintaining a red-black tree. All greedy algorithms, including Dayu, are executed using a single thread; all random-based algorithms are executed using 16 threads. Note that although random-based algorithms can be distributed to reach even higher speed, we find their speed is not the bottleneck anyway in our experiments. We also test the MILP algorithm with a state-of-the-art MILP solver Gurobi [33], but find it can only finish computation for a small-scale cluster; for a 3500-node cluster and only 2000 tasks, it cannot finish computation after 125 seconds and thus we do not report its results. When determining the rate of each task, we compare Dayu's iterative WSS (W) with deadline-based allocation (DA), which assigns a rate of $\frac{s_t}{T_{timeslot}}$ to task t so that a task can be finished in one timeslot.

5.1 Overall Performance

Evaluation on the Real-world Systems. Figure 5 shows the recovery times and the p90 latency of the foreground re-

System	Algorithm
Commons [3, 5–7]	Random placement
RAMClould [4]	Best-of-two-random
CAR [31]	Greedy1
PPR [32]	Greedy1
Mirador [34]	Greedy2
DH-HDFS [35]	MILP
Sparrow [36]	Best-of-two-random
Firmament [37]	MCMF

Table 2: State-of-the-art systems and their algorithms.

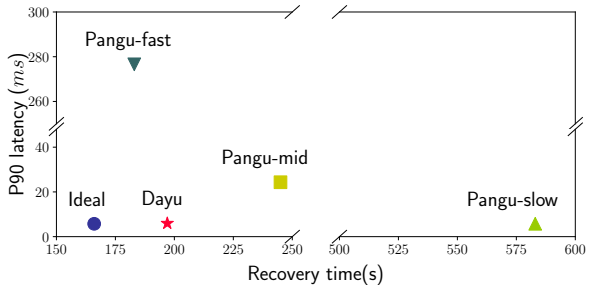


Figure 5: The recovery time and the p90 latency during recovery in real-world experiments

quests during recovery. In this test, we shutdown a server to create 15TB of data to recover, and approximately 990 surviving *ChunkServers* are responsible for recovery. For comparison, we add an "Ideal" entry in Figure 5, which estimates the optimal recovery time assuming all available bandwidth ($\alpha = 75\%$) can be utilized, and introduces no interference on the foreground traffic (i.e., the foreground latency is identical to the one without recovery).

As shown in the figure, Dayu achieves near-optimal recovery speed as well as low interference. First, Dayu is approaching the ideal recovery speed, as its recovery time is 1.19× longer than "Ideal": this is 2.96× and 1.24× faster than Pangu-slow (default) and Pangu-mid configurations respectively. Compared with the Pangu-fast configuration, although Dayu has a slightly slower recovery speed (0.93×), it introduces far less interference on the foreground traffic. Considering the interference of the recovery traffic on the foreground traffic, Dayu's p90 latency is only 1.04× longer than "Ideal". Pangu-slow has a slightly lower interference with its p90 latency nearly the same as "Ideal"; Pangu-mid and Pangu-fast create unacceptable interference as their p90 latencies are 4.23-48.14× higher than "Ideal". Due to the high interference to the foreground traffic, Pangu-mid and Pangu-fast are seldom used on production clusters. In summary, compared with the different settings in Pangu, Dayu achieves close-to-optimal recovery time and interference.

Evaluation on the Simulation Systems. Figure 6 shows the results of simulation experiments. Again, compared with other algorithms, Dayu achieves a good balance between recovery speed and interference.

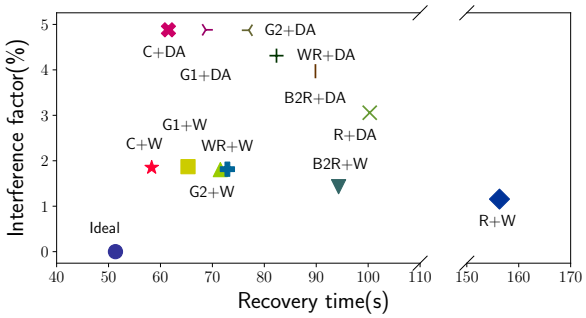


Figure 6: Data recovery in simulation (Dayu uses C+W)

In terms of recovery speed, Dayu's combination of dynamic convex hull node selection and iterative WSS (C+W) can achieve the shortest recovery time among all algorithms, which is 1.14× longer than the ideal recovery time. Compared to other algorithms, dynamic convex hull node selection achieves the fastest recovery speed, which is 1.12× faster than G1, the second best one. Note that though G1 is close to Dayu in this experiment, it does not scale well due to its high computation overhead (§5.2). For the greedy-based algorithms including Dayu, iterative WSS is slightly faster than deadline-based allocation, because the former can finish the last timeslot early when tasks are rare, while the latter must finish tasks at the end of the last timeslot. For random-based algorithms, such effect is unclear because the recovery speed is mainly determined by the selection of the sources and destinations.

In terms of interference on foreground, Dayu has acceptable interference factor (recall that a factor of 2% is small and a factor larger than 6.5% is unacceptable). With the same bandwidth allocation strategy, Dayu's node selection algorithm and other greedy algorithms have slightly larger interference than those random based algorithms, because greedy algorithms usually utilize more estimated available bandwidth. When the estimation of the foreground traffic has some errors, the interference will be slightly larger. With the same node selection algorithm, iterative WSS consistently brings lower interference than deadline-based allocation (DA).

5.2 Scalability

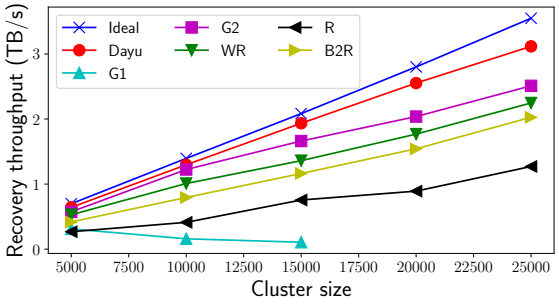


Figure 7: Dayu's scalability

We evaluate the scalability of Dayu beyond 3,500 nodes. To measure the full capability of different algorithms, we assume there are infinite number of recovery tasks and simulate how much data each algorithm can recover in 20 timeslots. As the scale of the simulated clusters are larger than our observed cluster, we randomly generate block placement based on the statistics from our collected traces; we randomly pick the foreground trace from one real node for one simulated node.

As shown in Figure 7, Dayu can scale to 25,000 nodes and till that point, the performance of Dayu is higher than all other algorithms. We do not test even larger scales because they are too far away from our target (10K nodes). Besides Dayu, all random algorithms scale pretty well, which is as expected, though their performance is not as good as Dayu. G1 does not scale to more than 5,000 nodes because of its high computation overhead. Note that as greedy algorithms, Dayu and G2 will eventually stop scaling at some point because of their centralized computation, but at least for the scale we target now and in the near future, the simulation shows that Dayu is fast enough and can provide better quality.

5.3 Effects of Individual Techniques

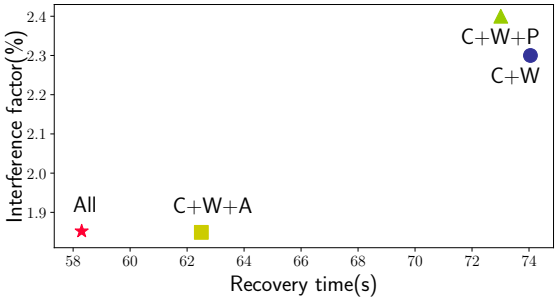


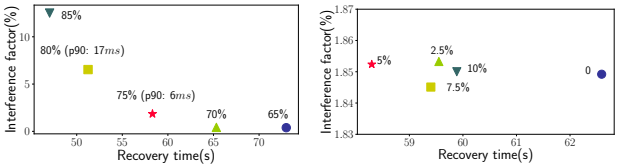
Figure 8: Effects of prioritizing underemployed node (P) and re-scheduling stragglers (A)

We further investigate the effects of prioritizing underemployed nodes (P) and re-scheduling stragglers (A) described in Section 4.2 and 4.4. We use Dayu equipped with convex hull node selection (C) and iterative WSS bandwidth allocation (W) as the baseline (C+W), which scans tasks with no prioritization and executes stragglers with the original plan (i.e. P and A are disabled). Note that in this baseline, Dayu is aware of those stragglers and will use their information to schedule the current timeslot but won't re-schedule stragglers.

Figure 8 presents Dayu's schedule results with and without P and A. As shown in the figure, the re-scheduling of stragglers is keen to the performance: compared with the baseline (C+W), re-scheduling stragglers (C+W+A) reduces recovery time by 15.6% and reduces the interference as well. Though prioritizing underemployed nodes has limited effect without re-scheduling stragglers, it accelerates the recovery speed by 7.2% when straggler re-scheduling is already equipped (compare "All" to the case C+W+A).

5.4 Impacts of Key Parameters

Finally, we measure the impacts of key parameters of Dayu. The first one is α in Equation 1, which controls the interference of recovery traffic on foreground traffic. Figure 9(a) plots the recovery time and interference factor as α increases from 65% to 85% with the step size of 5%. One can see that the larger the α, the shorter the recovery time but the larger the interference factor. In this figure, we further map some of these simulated interference factors to the p90 latencies from the real-world experiments, so that we can quantitatively understand the values of the simulated interference factors. Our decision to use the value 75% for α is mainly based on these p90 latencies from real-world experiments: with $\alpha = 75\%$, Dayu achieves close-to-optimal recovery time and p90 latency (§5.1); with $\alpha = 80\%$, although Dayu decreases recovery time by 9.1%, it almost triples the p90 latency of the foreground traffic.



(a) Tuning α (b) Tuning β

Figure 9: Effects of tuning α and β

The next parameter, β, denotes the ratio of selected nodes from two sorted lists when choosing underemployed *ChunkServers* in Section 4.2. We change β from 0% to 10% with the step size of 2.5%. As shown in Figure 9(b), the value of β has no significant impact on the interference factor and setting it to 5% achieves the lowest recovery time, which is why Dayu sets β to 5%.

Another important parameter is the length of a timeslot ($T_{timeslot}$), but since this parameter affects the overall overhead of Pangu, we were not allowed to change it in the production system and thus we were not able to record and analyze a trace with a different $T_{timeslot}$. In general, shorter $T_{timeslot}$ will benefit Dayu by allowing it to react to foreground fluctuation more quickly but will increase the overhead of Pangu.

6 Related Work

Data Recovery. Popular distributed filesystems such as GFS [3], HDFS [5], Cosmos [6], and Windows Azure Storage [7] use random node selection and static rate control for data recovery, same as Pangu. RAMCloud [4, 38] uses the best-of-two-random algorithm to select the source and destination for a recovery task. Constrained by the deterministic placement, consistent hashing based storage systems [17, 22, 39–41] have little flexibility to choose the destination. Our work shows that randomized algorithms may not have a good quality in a highly imbalanced environment.

Some works improve data recovery in erasure-coded storage, by accelerating recovery of one failed block [31, 42, 43] or designing new recovery-efficient codes [44–47]. Applying Dayu to erasure-coded storage is our future work.

Data migration. Data recovery can be viewed as a subtopic of data migration. A number of distributed filesystems [3, 5] trigger data migration with a simple strategy or even manually (e.g., running HDFS balancer [48]). Mirador [34] uses a priority queue to greedily migrate data objects according to pre-defined rules. However, experiments in [34] report it does not scale well due to its greedy algorithm. Curator [49] uses a reinforcement learning solution to determine when to start a migration task, but it does not choose sources and destinations for data migration. DH-HDFS [35] utilizes MILP solver to manage migration of large scale storage system, but for our problem, MILP is too slow.

Constrained data placement strategies. Besides consistent hashing based storage systems [17, 22, 39–41], there are other systems restricting the data placement. Facebook [50] modifies native HDFS to constrain the placement of block replicas into smaller node groups (i.e., with a smaller scatter width), reducing the probability of losing data due to simultaneous node failures. With a fixed scatter width, CopySets [51] and Tiered Replication [52] further try to minimize the number of the distinct copysets in the whole system to reduce the probability of data loss due to correlated node failures. We plan to investigate the applicability and effectiveness of Dayu on these strategies in the future.

Large scale scheduling. Many large-scale computation platforms need to schedule computation tasks, which is similar to schedule recovery tasks in Dayu. Most centralized schedulers [53, 54] have poor computation performance at a large scale, and thus distributed schedulers are widely discussed [36, 55, 56]. However, due to the lack of coordination and the latest state, these schedulers often fail to generate high quality decisions [37]. Firmament [37], a centralized scheduler, succeeds to scale to a 12500-node cluster [57] , but experiments in [37, 58] report it has limited scalability with massive short tasks, which is exactly our scenario (§2.2).

7 Conclusion

Our work shows that a centralized scheduler has better scheduling quality, especially in a dynamic and imbalanced environment; its weakness, i.e. relatively low speed compared to the decentralized schedulers, can be mitigated by different optimizations (e.g. timeslot-based scheduling, convex hull optimization, etc). As a result, it can support a reasonably large system we target.

8 Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd Sudarsun Kannan for his guidance during our camera-ready preparation. We also thank Tianyang Jiang for helpful discussions, and Alibaba Cloud for providing us the evaluation cluster. This work was supported by the National key R&D Program of China under Grant No. 2018YFB0203902, and the National Natural Science Foundation of China under Grant No. 61672315.

References

[1] Zujie Ren, Weisong Shi, Jian Wan, Feng Cao, and Jiangbin Lin. Realistic and scalable benchmarking cloud file systems: Practices and lessons from alicloud. *IEEE Transactions on Parallel and Distributed Systems*, 28(3272-3285):1, 2017.

[2] Backblaze. Backblaze Durability is 99.999999999% — And Why It Doesn't Matter. https://www.backblaze.com/blog/cloud-storage-durability/, 2018. Online; accessed 2018-12-25.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, volume 37. ACM, 2003.

[4] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 29–41. ACM, 2011.

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of 26th IEEE symposium on Mass storage systems and technologies (MSST'10)*, pages 1–10. IEEE, 2010.

[6] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment (VLDB'08)*, 1(2):1265–1276, 2008.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 143–157. ACM, 2011.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[10] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. BDS: a centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth European Conference on Computer Systems (Eurosys'18)*, pages 10:1–10:14. ACM, 2018.

[11] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3Sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth European Conference on Computer Systems (Eurosys'18)*, page 2. ACM, 2018.

[12] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys' 16)*, page 35. ACM, 2016.

[13] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, 2008.

[14] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'11)*, pages 98–109, New York, NY, USA, 2011. ACM.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[16] Apache. HDFS Federation. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html, 2018. Online; accessed 2018-04-16.

[17] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.

[18] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)*, volume 39, pages 51–62. ACM, 2009.

[19] Albert Greenberg, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO'08)*, pages 57–62. ACM, 2008.

[20] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'09)*, volume 39, pages 39–50. ACM, 2009.

[21] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'08)*, volume 38, pages 63–74. ACM, 2008.

[22] 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. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, volume 41, pages 205–220. ACM, 2007.

[23] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04)*, 51(1):107–113, 2004.

[25] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 1–14, Oakland, CA, 2018. USENIX Association.

[26] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[27] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in Situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 1–16, Carlsbad, CA, 2018. USENIX Association.

[28] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'13)*, volume 43, pages 231–242. ACM, 2013.

[29] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, Berkeley, CA, USA, 2010. USENIX Association.

[30] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'12)*, pages 127–138. ACM, 2012.

[31] Zhirong Shen, Jiwu Shu, and Patrick PC Lee. Reconsidering single failure recovery in clustered file systems. In *Proceedings of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 323–334. IEEE, 2016.

[32] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, page 30. ACM, 2016.

[33] Gurobi. Gurobi 8.0. URL:http://www.gurobi.com, 2018. Online; accessed 2018-12-25.

[34] Jake Wires and Andrew Warfield. Mirador: An active control plane for datacenter storage. In *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 213–228, Santa Clara, CA, 2017. USENIX Association.

[35] Pulkit A. Misra, Inigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *Proceedings of 2017 USENIX Annual Technical Conference (ATC'17)*, pages 799–811, Santa Clara, CA, 2017. USENIX Association.

[36] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[37] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 99–115, Savannah, GA, 2016. USENIX Association.

[38] Ryan Scott Stutsman. *Durability and crash recovery in distributed in-memory storage systems*. PhD thesis, Stanford University, 2013.

[39] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[40] Gluster. GlusterFS documention. URL:https://docs.gluster.org/en/latest/, 2018. Online; accessed 2018-12-25.

[41] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the ACM SIGCOMM 2001 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'01)*, 31(4):149–160, 2001.

[42] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*, pages 567–579, 2017.

[43] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, page 30. ACM, 2016.

[44] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[45] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 81–94, 2015.

[46] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST'18)*, volume 2018, pages 139–154, 2018.

[47] Min Ye and Alexander Barg. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory*, 63(10):6307–6317, 2017.

[48] Apache. HDFS Balancer Command. URL:https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html, 2019. Online; accessed 2019-01-10.

[49] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 51–66, Boston, MA, 2017. USENIX Association.

[50] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*, pages 1071–1080. ACM, 2011.

[51] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 37–48, 2013.

[52] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 31–43, 2015.

[53] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew 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. ACM, 2009.

[54] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[55] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[56] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, pages 499–510, Santa Clara, CA, 2015. USENIX Association.

[57] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC'12)*, page 7. ACM, 2012.

[58] Ionel Corneliu Gog. *Flexible and efficient computation in large data centres*. PhD thesis, University of Cambridge, 2018.