Topic preview: Scheduling

Siddhartha Sen

Microsoft Research NYC

Scheduling

- Tells you how and when to assign tasks to processors
  - Task = anything you can execute
  - Processor = anything you can execute it on

Scheduling

- Tells you how and when to assign tasks to processors
  - Task = runnable thread, web request, MapReduce job
  - Processors = CPU core, service backend, cluster machine

web request $\rightarrow$ service backend $\rightarrow$ worker thread $\rightarrow$ kernel thread $\rightarrow$ CPU core

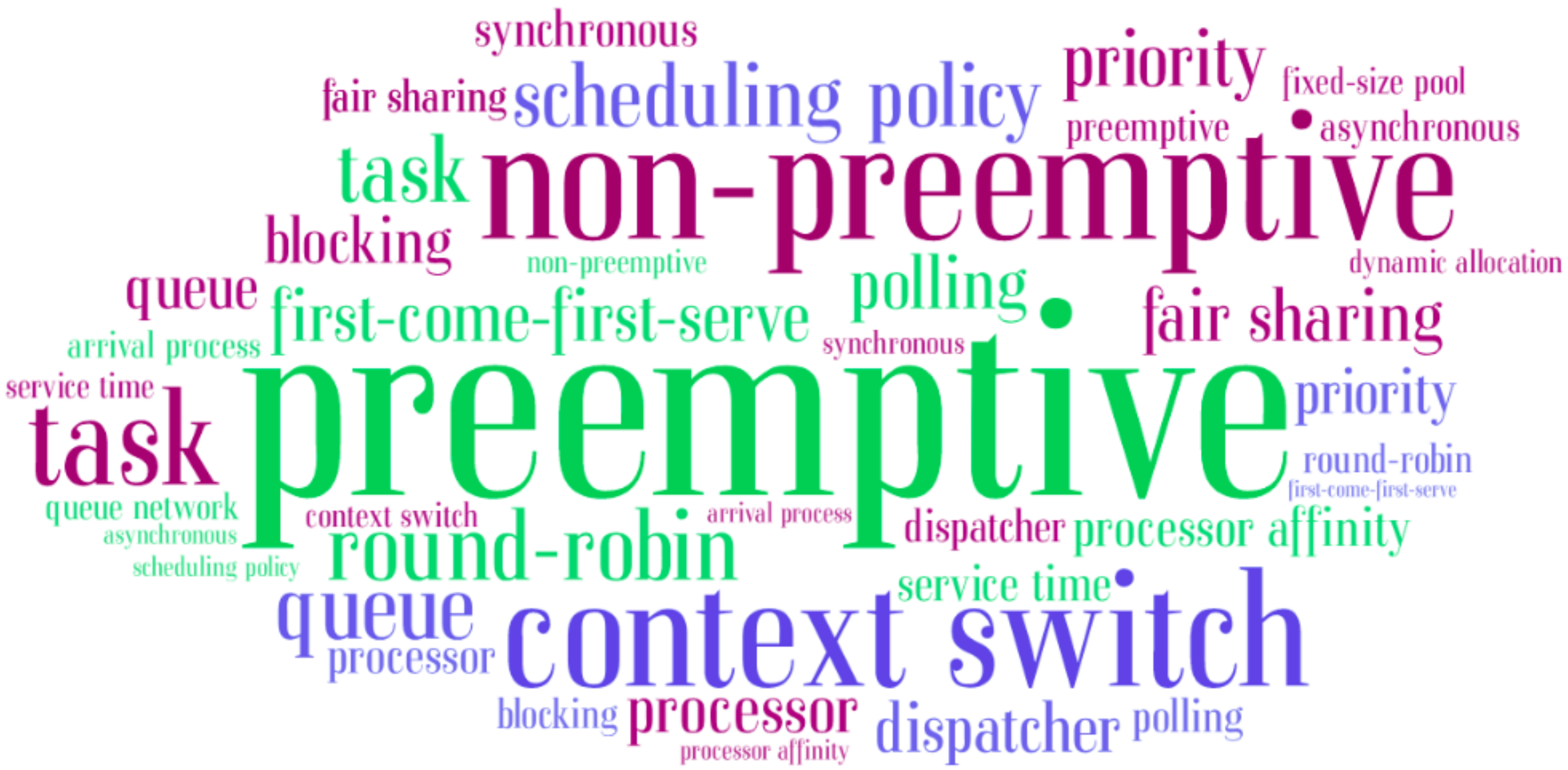
Scheduling

- Tells you how and when to assign tasks to processors
  - Task = runnable thread, web request, MapReduce job
  - Processors = CPU core, service backend, cluster machine

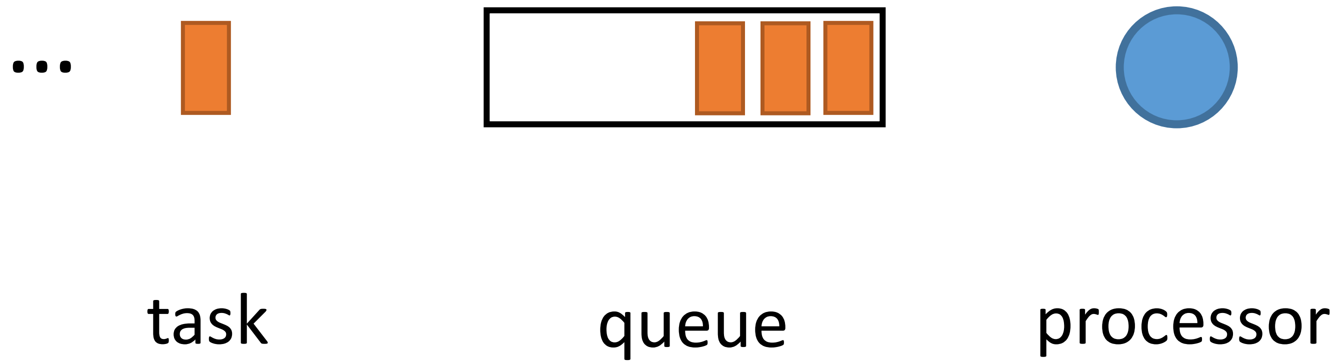
- Many different settings
  - Within a machine, across machines
  - Single-tiered, multi-tiered
  - Online, offline

- Several different objectives
  - Latency, throughput
  - Priority, fairness

All you need to know



Theoretical view

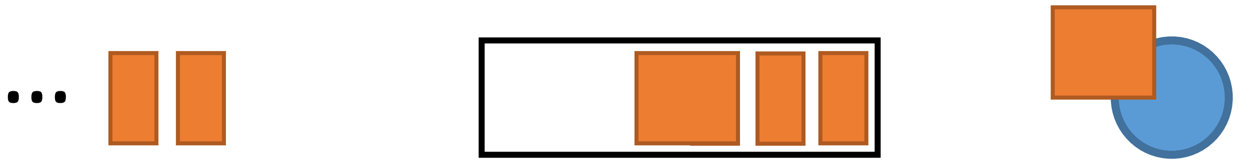


Theoretical view

Theoretical view

arrival process
(e.g., Poisson)

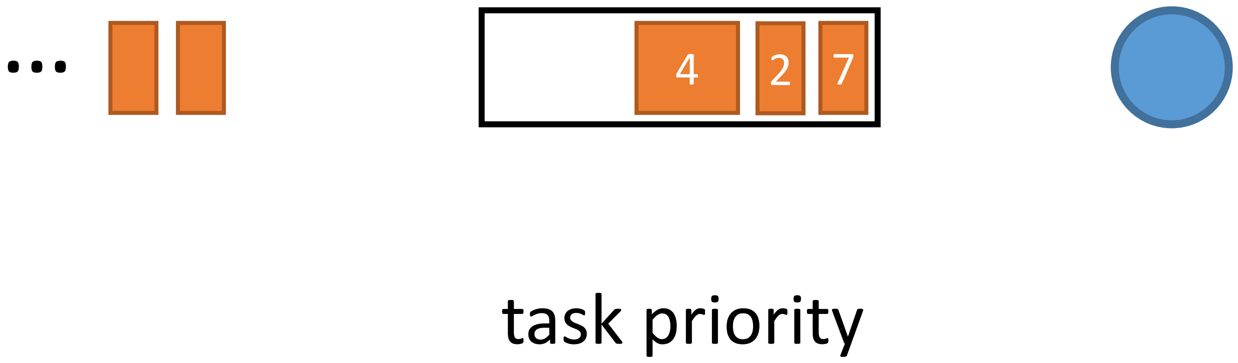
Theoretical view



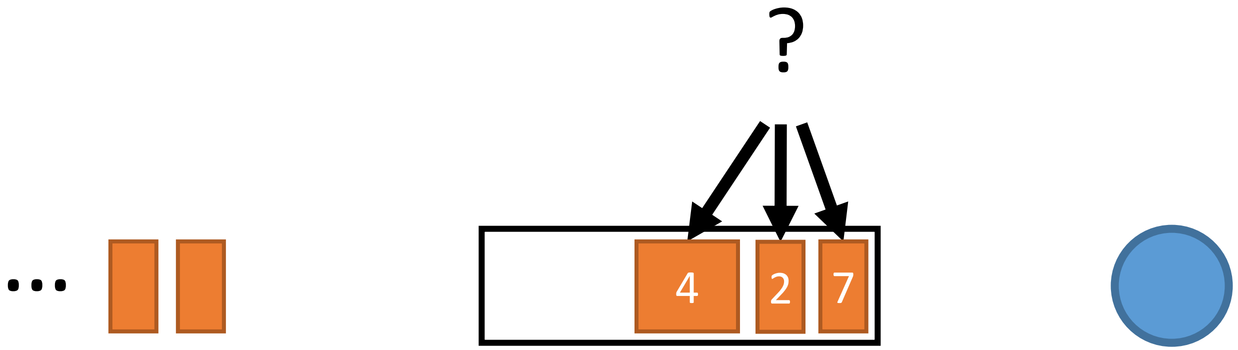
task size

service time
(e.g., exponential dist.)

Theoretical view



Theoretical view

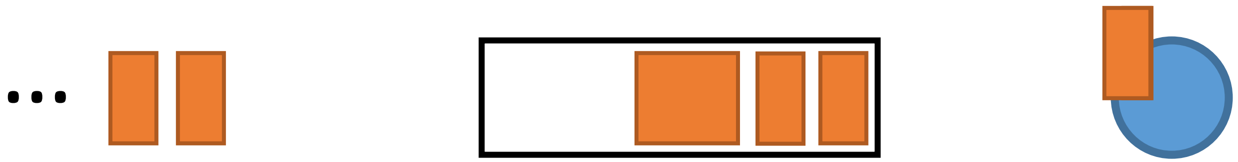


scheduling policy

- first-come-first-served
- shortest-task-first
- priority

Theoretical view

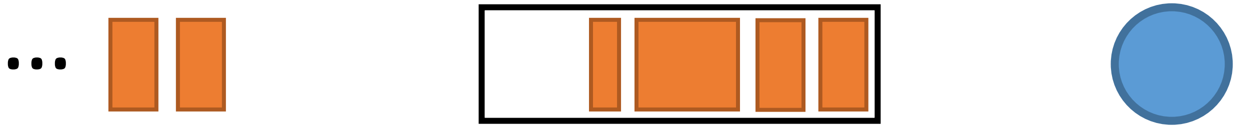
context switch
(by dispatcher)



fair sharing

preemptive?

Theoretical view

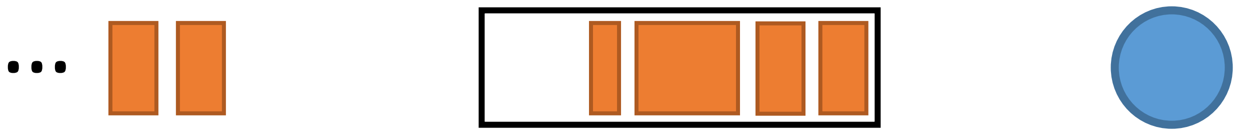


scheduling policy

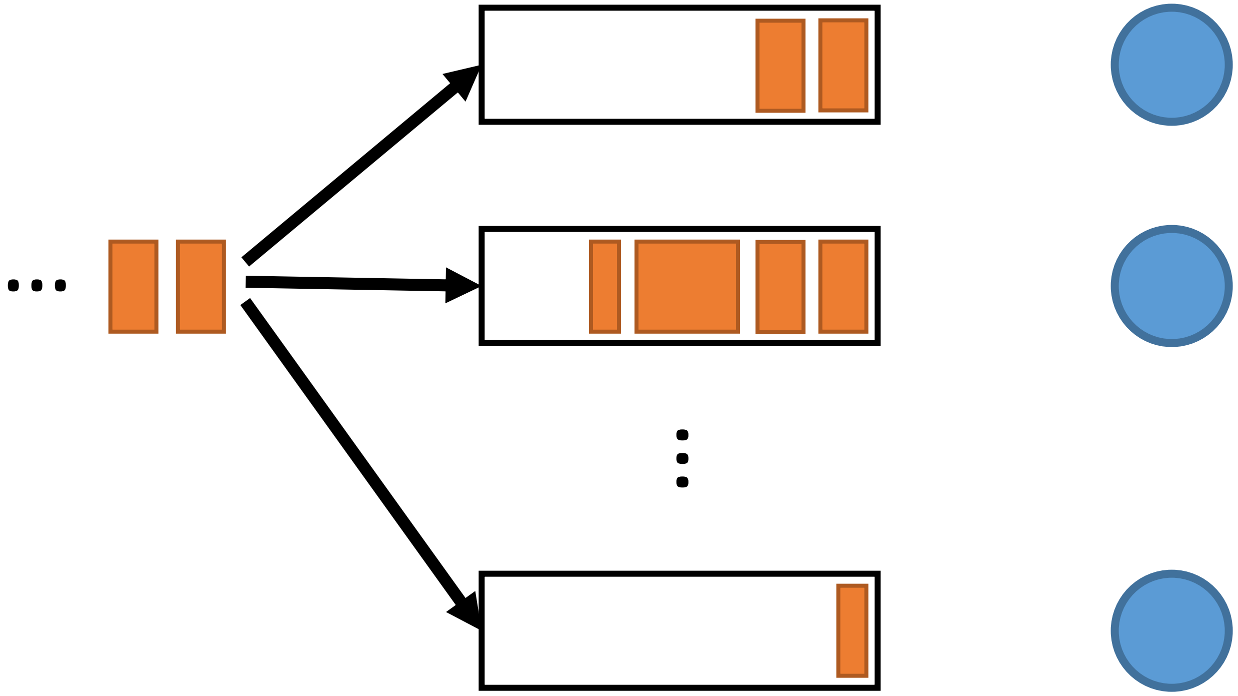
- round-robin (time quantum)
- shortest-remaining-time-first*
- priority

* related to work on progress indicators

Theoretical view



Theoretical view



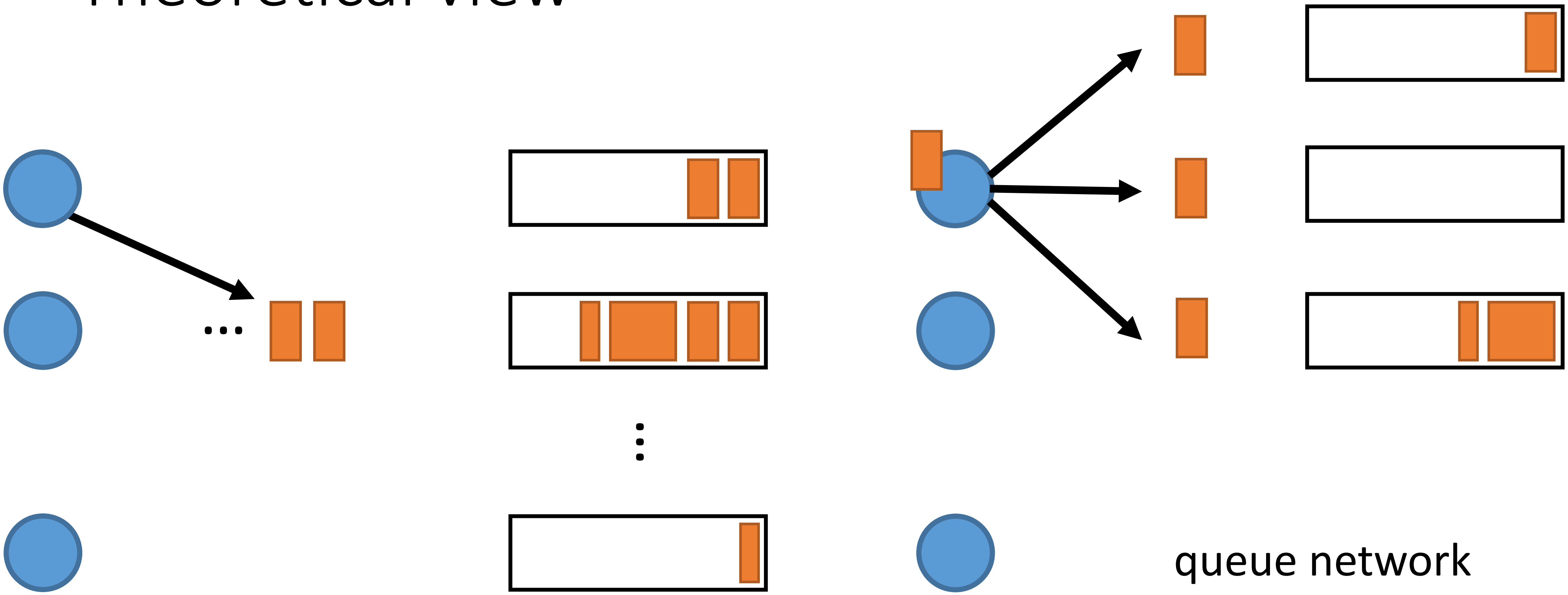
fixed-sized pool/
dynamic allocation

blocking/
polling

synchronous/
asynchronous

processor affinity
(cache locality, NUMA)

Theoretical view



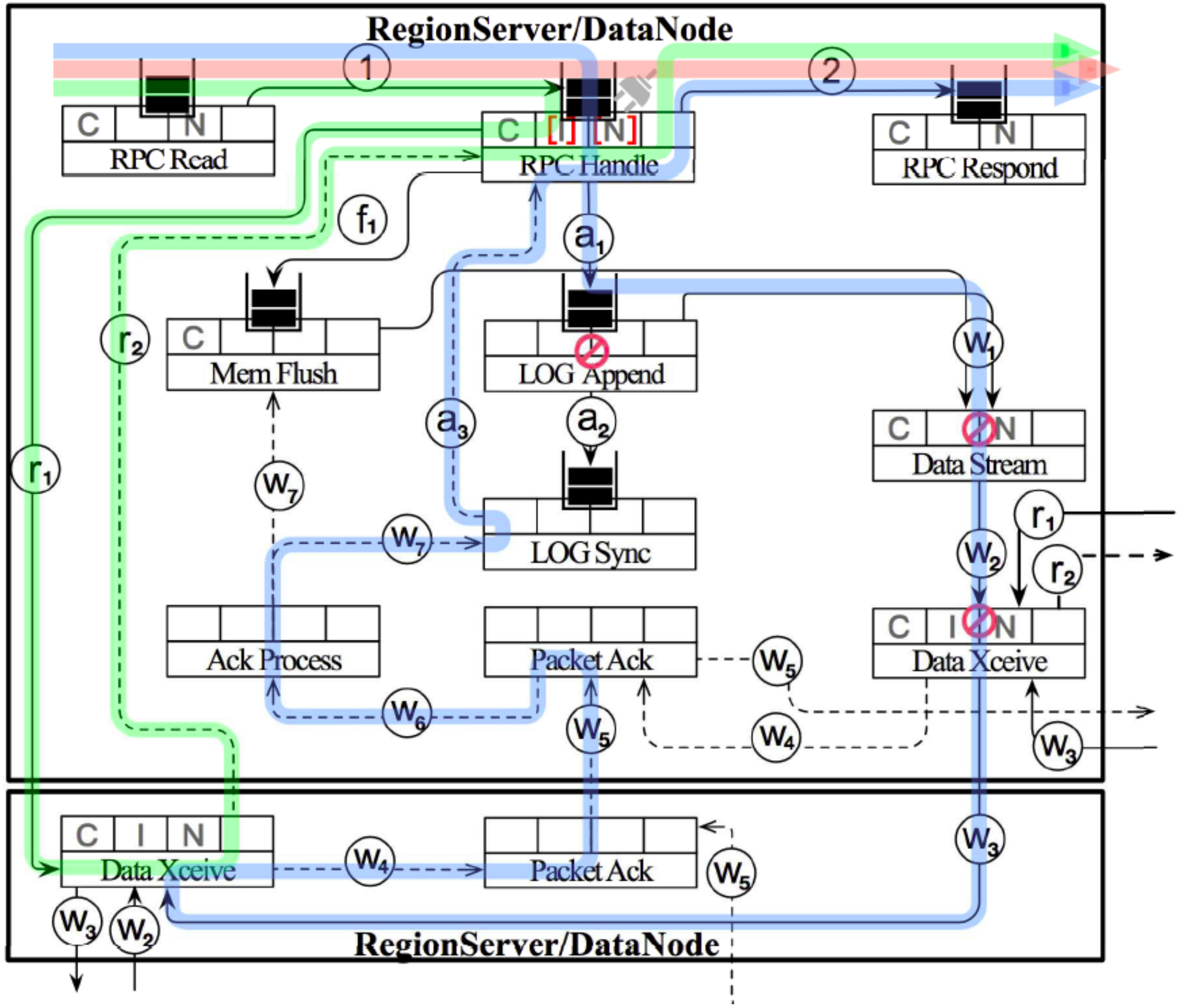
Takeaways?

- Lots of terms, organized by queueing theory

- Theory makes simplifying assumptions
  - Example: Shortest-task-first is “optimal” because it yields lowest average latency
- In practice no single scheduling policy is best
  - Example: Shortest-task-first needs accurate a priori estimate of task size
    Also, what about tail latency?

- There are many practical issues:
  - How is work dispatched to processors?
  - How do processors wait for work?
  - What is a “processor”? How is it allocated?
    - E.g.: kernel thread managed by Linux scheduler

These practical issues matter

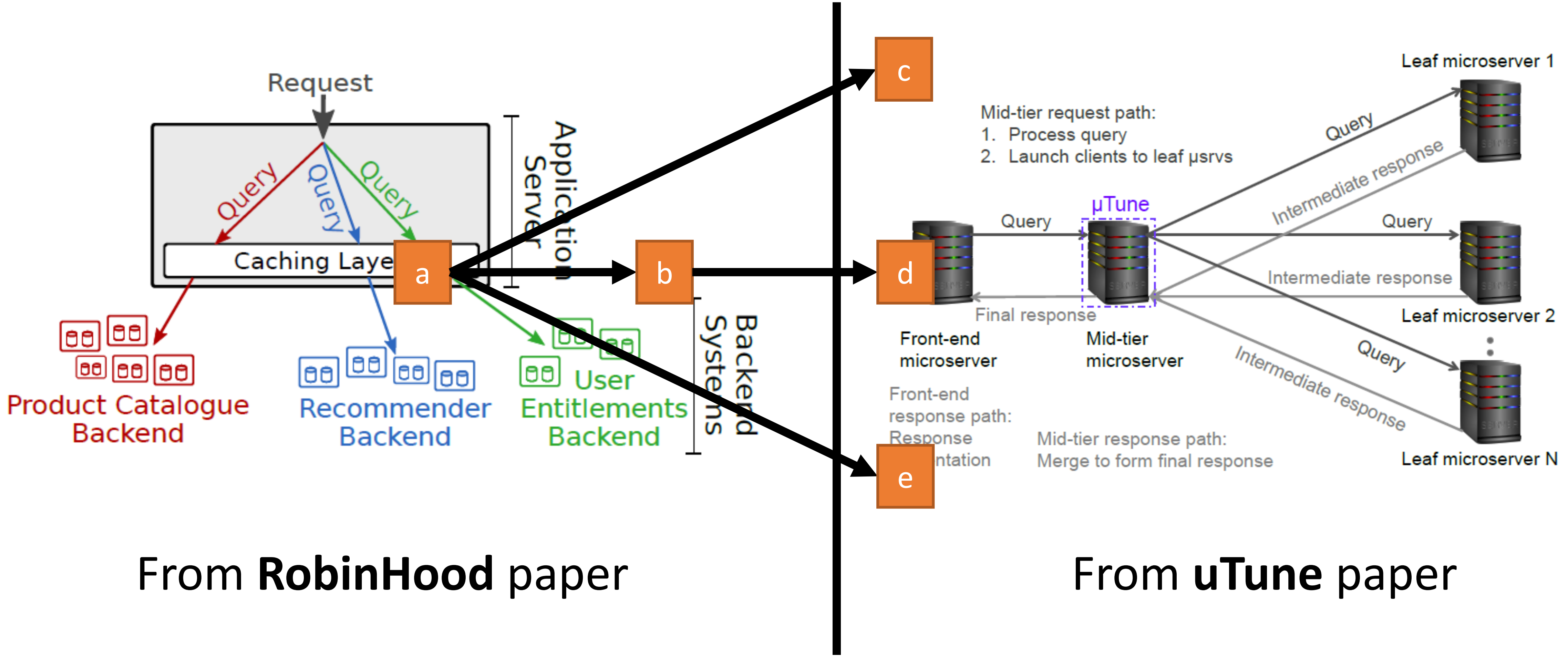


From **TAM** paper

Session focuses on practical issues

- **Arachne:** Scheduling short tasks on a virtual resource (threads) leads to poor utilization/performance
- **TAM:** Several common problems (e.g., hidden contention, ordering constraints) prevent systems from being schedulable
- **uTune:** Right threading model for multi-tiered microservices depends critically on load
- **RobinHood:** Tail latency depends on request structure and latency of backend services out of our control

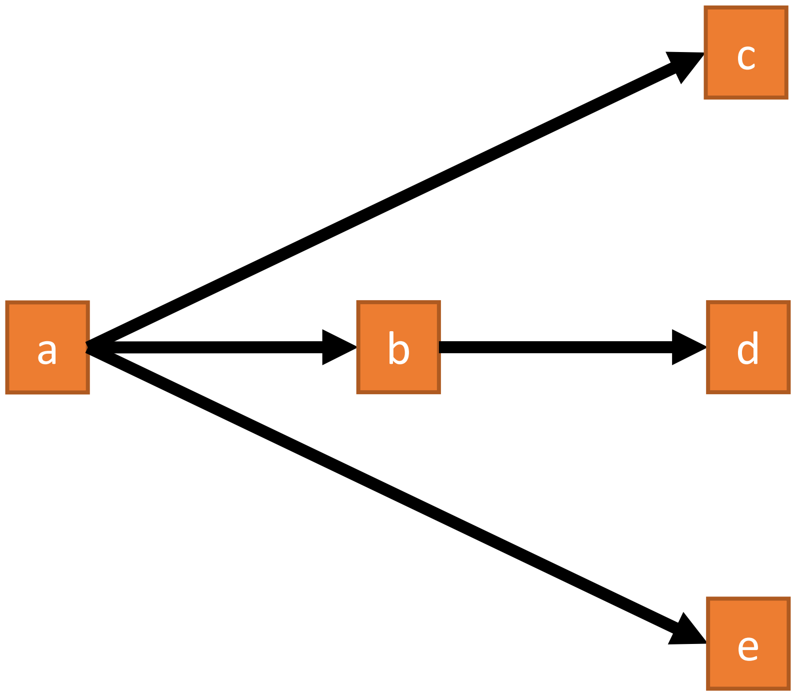
Running example: request in multi-tier system



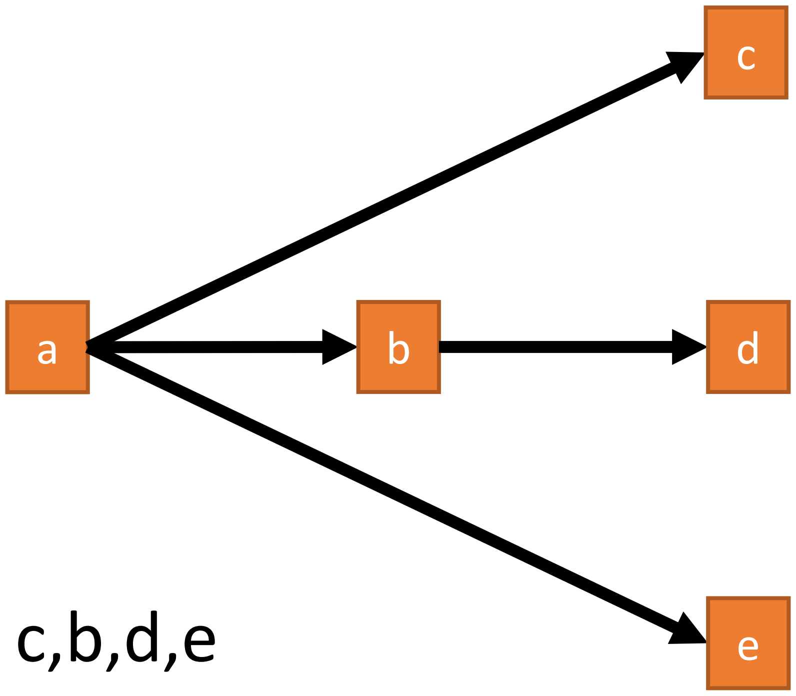
From **RobinHood** paper

From **uTune** paper

Running example: request in multi-tier system



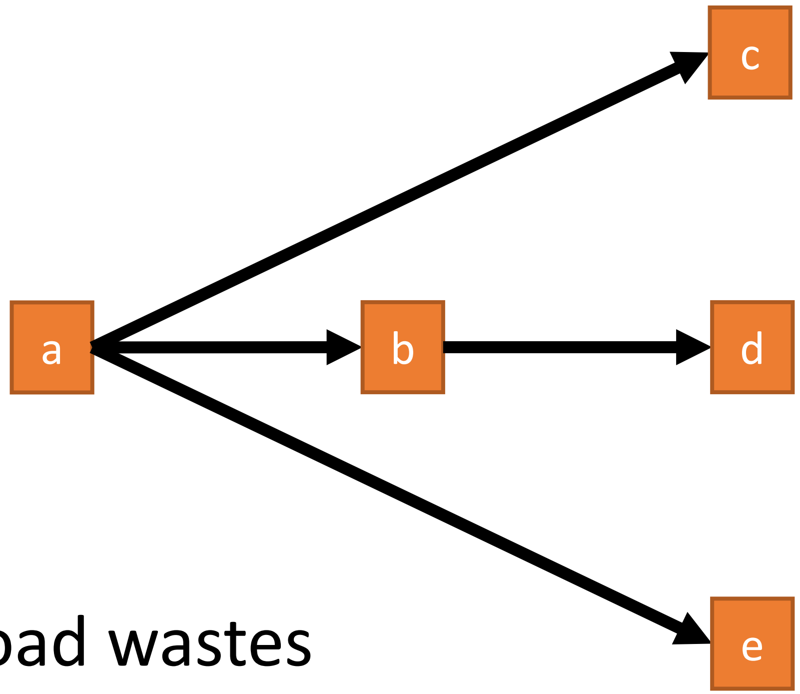
Problem observed by **Arachne**



Busy-waiting for c,b,d,e wastes CPU

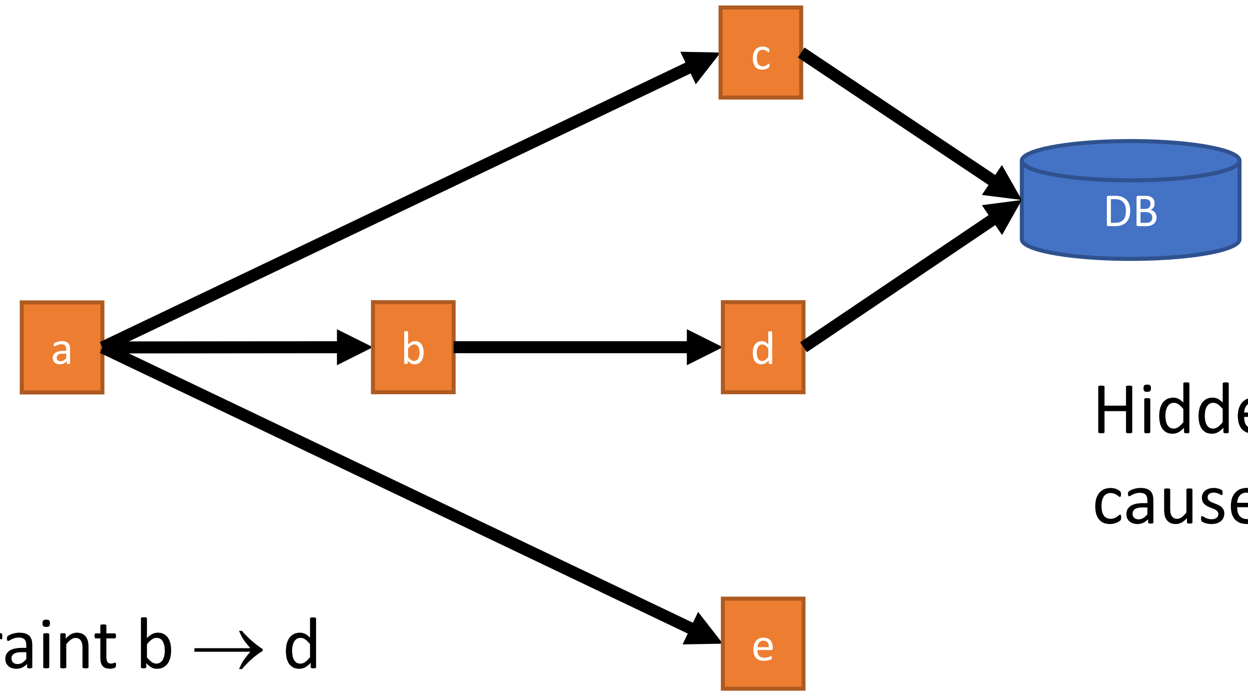
(e.g., write replication in RAMCloud)

Problem observed by **uTune**



Polling at high load wastes CPU; blocking at low load causes expensive wakeups

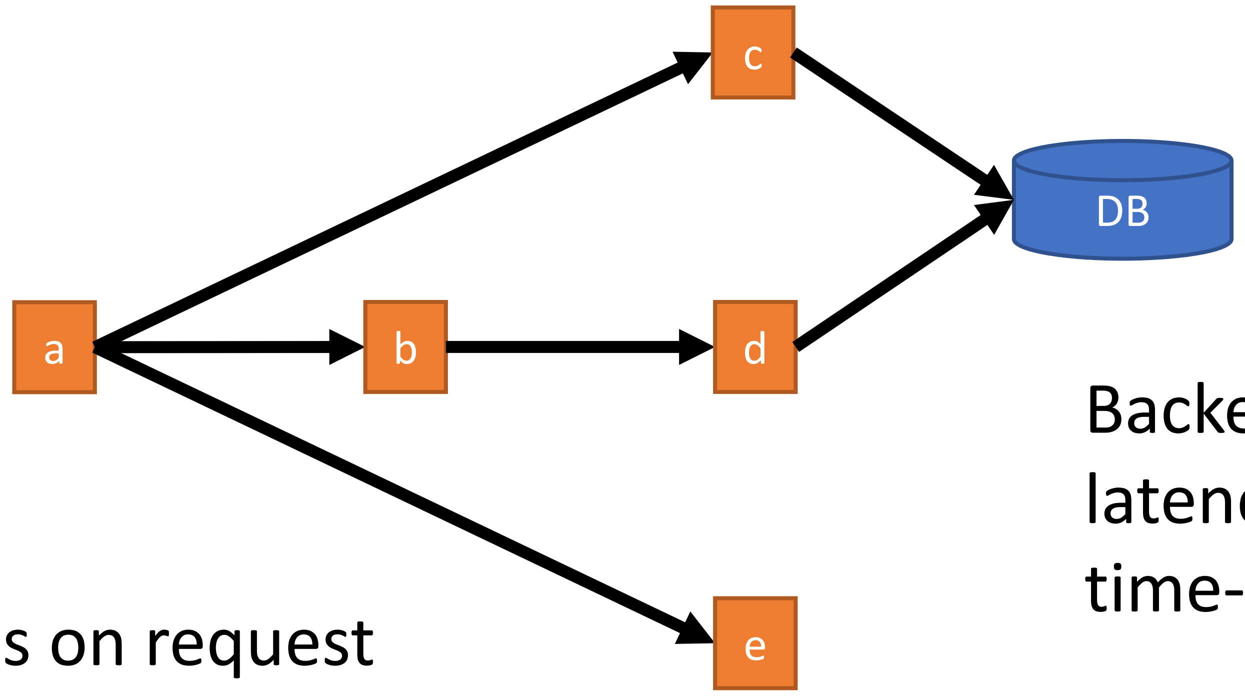
Problem observed by TAM



Hidden contention causes unfairness

Ordering constraint b $\rightarrow$ d reduces scheduling options

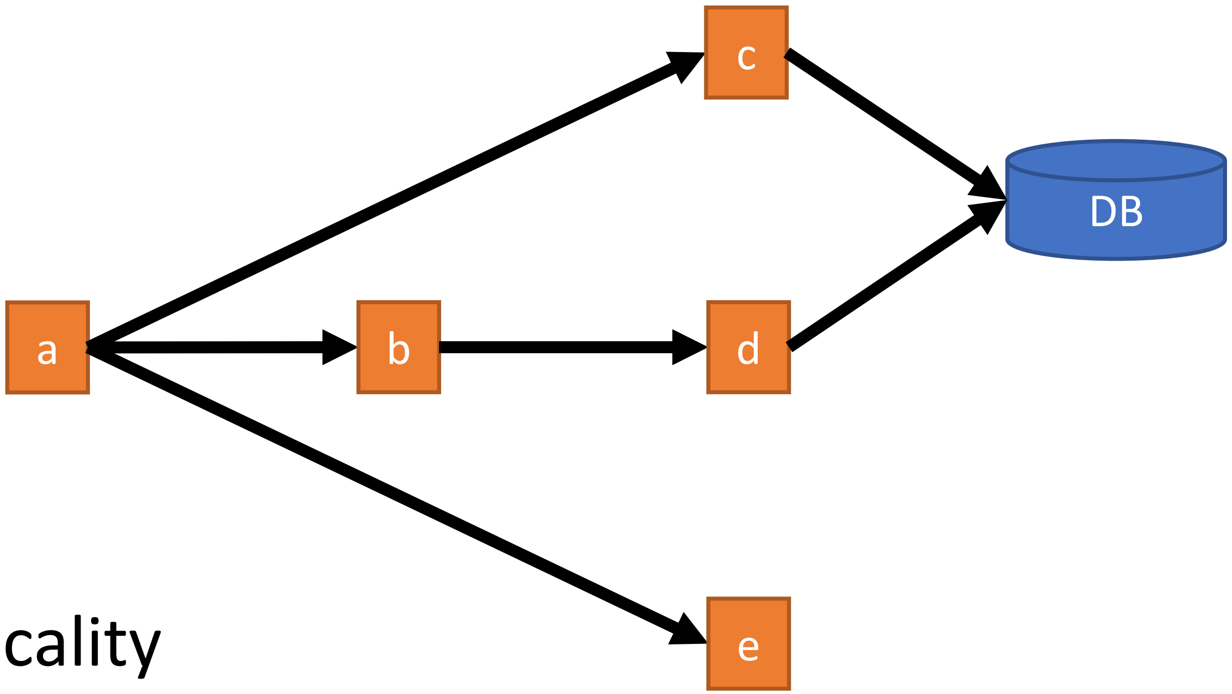
Problem observed by **RobinHood**



Backend causing tail latency is unpredictable, time-varying

Latency depends on request structure – i.e., max(b+d,c,e) – which is typically ignored

Problem observed by **all papers**



Cache misses/locality affect tail latencies!

Solution: **Arachne**

- Adaptively and exclusively assign CPU cores to applications; let app schedule user threads
- Designed to minimize cache misses (e.g., no ready queues)
- Improved Memcached, RAMCloud

Arachne: Core-Aware Thread Management

Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout
{hq6,qianli,jspeiser,kraftp,ouster}@cs.stanford.edu
Stanford University

Abstract

Arachne is a new user-level implementation of threads that provides both low latency and high throughput for applications with extremely short-lived threads (only a few microseconds). Arachne is *core-aware*: each application determines how many cores it needs, based on its load; it always knows exactly which cores it has been allocated, and it controls the placement of its threads on those cores. A central core arbiter allocates cores between applications. Adding Arachne to memcached improved SLO-compliant throughput by 37%, reduced tail latency by more than 10x, and allowed memcached to coexist with background applications with almost no performance impact. Adding Arachne to the RAMCloud storage system increased its write throughput by more than 2.5x. The Arachne threading library is optimized to minimize cache misses; it can initiate a new user thread on a different core (with load balancing) in 320 ns. Arachne is implemented entirely at user level on Linux; no kernel modifications are needed.

1 Introduction

Advances in networking and storage technologies have made it possible for datacenter services to operate at exceptionally low latencies [5]. As a result, a variety of low-latency services have been developed in recent years, including FaRM [11], Memcached [23], MICA [20], RAMCloud [30], and Redis [34]. They offer end-to-end response times as low as 5 µs for clients within the same datacenter and they have internal request service times as low as 1–2 µs. These systems employ a variety of new techniques to achieve their low latency, including polling instead of interrupts, kernel bypass, and run to completion [6, 31].

However, it is difficult to construct services that provide both low latency and high throughput. Techniques for achieving low latency, such as reserving cores for peak throughput or using polling instead of interrupts, waste resources. Multi-level services, in which servicing one request may require nested requests to other servers (such as for replication), create additional opportunities for resource underutilization, particularly if they use polling to reduce latency. Background activities within a service, such as garbage collection, either require additional reserved (and hence underutilized) resources, or risk interference with foreground request servicing. Ideally, it should be possible to colocate throughput-oriented services such as MapReduce [10] or video processing [22] with low-latency services, such that resources are fully occupied by the throughput-oriented services when not needed by the low-latency services. However, this is rarely attempted in practice because it impacts the performance of the latency-sensitive services.

One of the reasons it is difficult to combine low latency and high throughput is that applications must manage their parallelism with a virtual resource (threads); they cannot tell the operating system how many physical resources (cores) they need, and they do not know which cores have been allocated for their use. As a result, applications cannot adjust their internal parallelism to match the resources available to them, and they cannot use application-specific knowledge to optimize their use of resources. This can lead to both under-utilization and over-commitment of cores, which results in poor resource utilization and/or suboptimal performance. The only recourse for applications is to pin threads to cores; this results in under-utilization of cores within the application and does not prevent other applications from being scheduled onto the same cores.

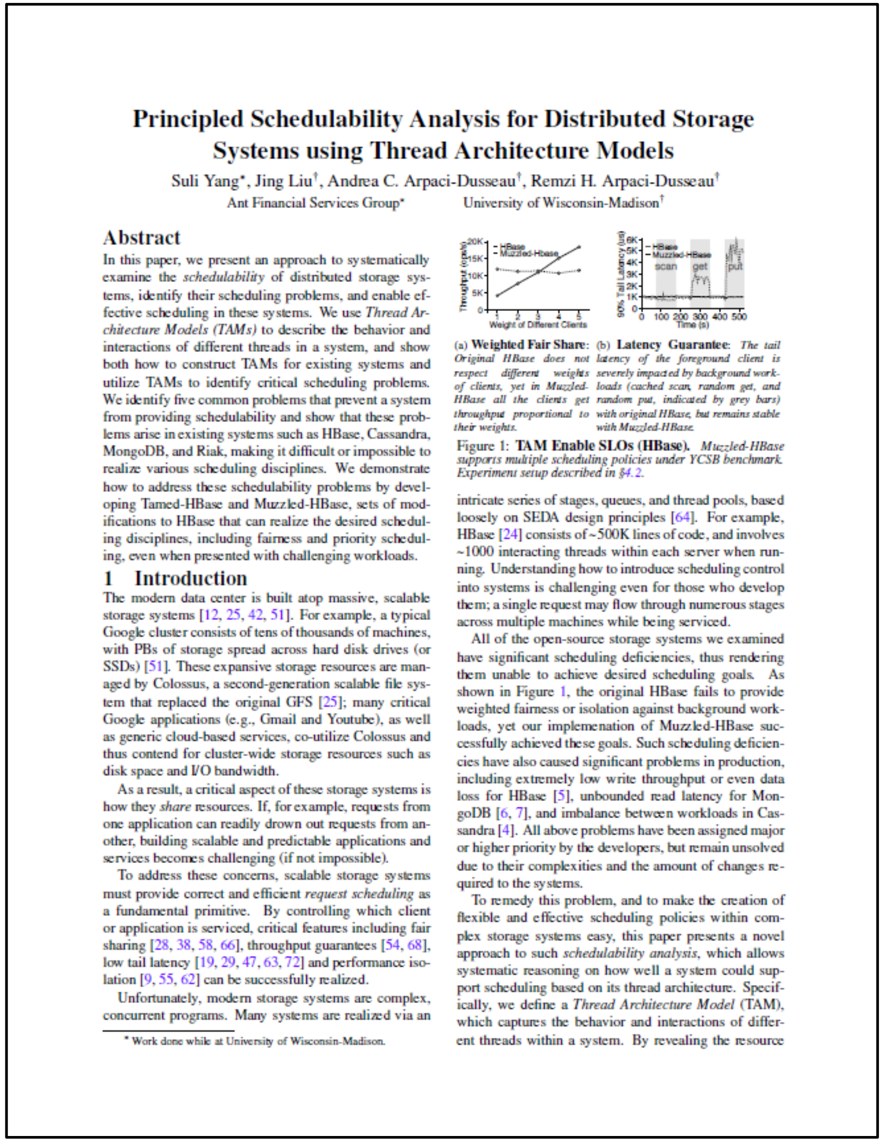
Arachne is a thread management system that solves these problems by giving applications visibility into the physical resources they are using. We call this approach *core-aware thread management*. In Arachne, application threads are managed entirely at user level; they are not visible to the operating system. Applications negotiate with the system over cores, not threads. Cores are allocated for the exclusive use of individual applications and remain allocated to an application for long intervals (tens of milliseconds). Each application always knows exactly which cores it has been allocated and it decides how to schedule application threads on cores. A *core arbiter* decides how many cores to allocate to each application, and adjusts the allocations in response to changing application requirements.

User-level thread management systems have been implemented many times in the past [39, 14, 4] and the basic features of Arachne were prototyped in the early 1990s in the form of scheduler activations [2]. Arachne is novel in the following ways:

- Arachne contains mechanisms to estimate the number of cores needed by an application as it runs.
- Arachne allows each application to define a *core policy*, which determines at runtime how many cores the application needs and how threads are placed on the available cores.
- The Arachne runtime was designed to minimize cache misses. It uses a novel representation of scheduling

Solution: **TAM**

- Automatically generate and visualize Thread Architecture Model (TAM) of system
- Use TAM to identify (and address) common problems preventing schedulability
- Improved HBase



Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models

Suli Yang*, Jing Liu†, Andrea C. Arpaci-Dusseau†, Remzi H. Arpaci-Dusseau†
Ant Financial Services Group* University of Wisconsin-Madison†

Abstract

In this paper, we present an approach to systematically examine the *schedulability* of distributed storage systems, identify their scheduling problems, and enable effective scheduling in these systems. We use *Thread Architecture Models (TAMs)* to describe the behavior and interactions of different threads in a system, and show both how to construct TAMs for existing systems and utilize TAMs to identify critical scheduling problems. We identify five common problems that prevent a system from providing schedulability and show that these problems arise in existing systems such as HBase, Cassandra, MongoDB, and Riak, making it difficult or impossible to realize various scheduling disciplines. We demonstrate how to address these schedulability problems by developing Tamed-HBase and Muzzled-HBase, sets of modifications to HBase that can realize the desired scheduling disciplines, including fairness and priority scheduling, even when presented with challenging workloads.

1 Introduction

The modern data center is built atop massive, scalable storage systems [12, 25, 42, 51]. For example, a typical Google cluster consists of tens of thousands of machines, with PBs of storage spread across hard disk drives (or SSDs) [51]. These expansive storage resources are managed by Colossus, a second-generation scalable file system that replaced the original GFS [25]; many critical Google applications (e.g., Gmail and Youtube), as well as generic cloud-based services, co-utilize Colossus and thus contend for cluster-wide storage resources such as disk space and I/O bandwidth.

As a result, a critical aspect of these storage systems is how they *share* resources. If, for example, requests from one application can readily drown out requests from another, building scalable and predictable applications and services becomes challenging (if not impossible).

To address these concerns, scalable storage systems must provide correct and efficient *request scheduling* as a fundamental primitive. By controlling which client or application is serviced, critical features including fair sharing [28, 38, 58, 66], throughput guarantees [54, 68], low tail latency [19, 29, 47, 63, 72] and performance isolation [9, 55, 62] can be successfully realized.

Unfortunately, modern storage systems are complex, concurrent programs. Many systems are realized via an

(a) **Weighted Fair Share**: *Original HBase does not respect different weights of clients, yet in Muzzled-HBase all the clients get throughput proportional to their weights.*

(b) **Latency Guarantee**: *The tail latency of the foreground client is severely impacted by background workloads (cached scan, random get, and random put, indicated by grey bars) with original HBase, but remains stable with Muzzled-HBase.*

Figure 1: **TAM Enable SLOs (HBase).** *Muzzled-HBase supports multiple scheduling policies under YCSB benchmark. Experiment setup described in §4.2.*

intricate series of stages, queues, and thread pools, based loosely on SEDA design principles [64]. For example, HBase [24] consists of ~500K lines of code, and involves ~1000 interacting threads within each server when running. Understanding how to introduce scheduling control into systems is challenging even for those who develop them; a single request may flow through numerous stages across multiple machines while being serviced.

All of the open-source storage systems we examined have significant scheduling deficiencies, thus rendering them unable to achieve desired scheduling goals. As shown in Figure 1, the original HBase fails to provide weighted fairness or isolation against background workloads, yet our implemenation of Muzzled-HBase successfully achieved these goals. Such scheduling deficiencies have also caused significant problems in production, including extremely low write throughput or even data loss for HBase [5], unbounded read latency for MongoDB [6, 7], and imbalance between workloads in Cassandra [4]. All above problems have been assigned major or higher priority by the developers, but remain unsolved due to their complexities and the amount of changes required to the systems.

To remedy this problem, and to make the creation of flexible and effective scheduling policies within complex storage systems easy, this paper presents a novel approach to such *schedulability analysis*, which allows systematic reasoning on how well a system could support scheduling based on its thread architecture. Specifically, we define a *Thread Architecture Model* (TAM), which captures the behavior and interactions of different threads within a system. By revealing the resource

* Work done while at University of Wisconsin-Madison.

Solution: uTune

- Taxonomize threading models to understand effect on tail latency
- Dynamically switch threading model based on load
- Improved services from uSuite

μTune: Auto-Tuned Threading for OLDI Microservices

Akshitha Sriraman Thomas F. Wenisch

University of Michigan

akshitha@umich.edu, twenisch@umich.edu

ABSTRACT

Modern On-Line Data Intensive (OLDI) applications have evolved from monolithic systems to instead comprise numerous, distributed microservices interacting via Remote Procedure Calls (RPCs). Microservices face sub-millisecond (sub-ms) RPC latency goals, much tighter than their monolithic counterparts that must meet ≥ 100 ms latency targets. Sub-ms–scale threading and concurrency design effects that were once insignificant for such monolithic services can now come to dominate in the sub-ms–scale microservice regime. We investigate how threading design critically impacts microservice tail latency by developing a *taxonomy of threading models*—a structured understanding of the implications of how microservices manage concurrency and interact with RPC interfaces under wide-ranging loads. We develop *µTune*, a system that has two features: (1) a novel framework that abstracts threading model implementation from application code, and (2) an automatic load adaptation system that curtails microservice tail latency by exploiting inherent latency trade-offs revealed in our taxonomy to transition among threading models. We study *µTune* in the context of four OLDI applications to demonstrate up to 1.9× tail latency improvement over static threading choices and state-of-the-art adaptation techniques.

1 Introduction

On-Line Data Intensive (OLDI) applications, such as web search, advertising, and online retail, form a major fraction of data center applications [113]. Meeting soft real-time deadlines in the form of Service Level Objectives (SLOs) determines end-user experience [21, 46, 55, 95] and is of paramount importance. Whereas OLDI applications once had largely monolithic software architectures [50], modern OLDI applications comprise numerous, distributed microservices [66, 90, 116] like HTTP connection termination, key-value serving [72], query rewriting [48], click tracking, access-control management, protocol routing [25], etc. Several companies, such as Amazon [6], Netflix [1], Gilt [37], LinkedIn [17], and SoundCloud [9], have adopted microservice architectures to improve OLDI development and scalability [144]. These microservices are composed via standardized Remote Procedure Call (RPC) interfaces, such as Google's Stubby and gRPC [18] or Facebook/Apache's Thrift [14].

Whereas monolithic applications face ≥ 100 ms tail (99^{th}+%) latency SLOs (e.g.,~300 ms for web search [126, 133, 142, 150]), microservices must often achieve sub-ms (e.g., ~100 µs for protocol routing [151]) tail latencies as many microservices must be invoked serially to serve a user's query. For example, a Facebook news feed service [79] query may flow through a serial pipeline of many microservices, such as (1) `Sigma` [15]: a spam filter, (2) `McRouter` [118]: a protocol router, (3) `Tao` [56]: a distributed social graph data store, (4) `MyRocks` [29]: a user database, etc., thereby placing tight sub-ms latency SLOs on individual microservices. We expect continued growth in OLDI data sets and applications to require composition of ever more microservices with increasingly complex interactions. Hence, the pressure for better microservice latency SLOs continually mounts.

Threading and concurrency design have been shown to critically affect OLDI response latency [76, 148]. But, prior works [71] focus on monolithic services, which typically have ≥ 100 ms tail SLOs [111]. Hence, sub-ms–scale OS and network overheads (e.g., a context switch cost of 5-20 µs [101, 141]) are often insignificant for monolithic services. However, sub-ms–scale microservices differ intrinsically: spurious context switches, network/RPC protocol delays, inept thread wakeups, or lock contention can dominate microservice latency distributions [39]. For example, even a single 20µs spurious context switch implies a 20% latency penalty for a request to a 100 µs SLO protocol routing microservice [151]. Hence, prior conclusions must be revisited for the microservice regime [49].

In this paper, we study how threading design affects mi-

Solution: **RobinHood**

- Explicitly identify backends contributing to tail latency
- Dynamically reallocate cache resources from cache-rich to cache-poor (those causing tail)
- Improved OneRF production system at Microsoft

RobinHood: Tail Latency-Aware Caching —
Dynamically Reallocating from Cache-Rich to Cache-Poor

Daniel S. Berger[1], Benjamin Berg[1], Timothy Zhu[2], Mor Harchol-Balter[1], and Siddhartha Sen[3]

[1]Carnegie Mellon University — [2]Penn State — [3]Microsoft Research

Abstract

Tail latency is of great importance in user-facing web services. However, maintaining low tail latency is challenging, because a single request to a web application server results in multiple queries to complex, diverse backend services (databases, recommender systems, ad systems, etc.). A *request* is not complete until *all of its queries* have completed. We analyze a Microsoft production system and find that backend query latencies vary by more than two orders of magnitude *across backends* and *over time*, resulting in high request tail latencies.

We propose a novel solution for maintaining low request tail latency: repurpose existing caches to mitigate the effects of backend latency variability, rather than just caching popular data. Our solution, RobinHood, dynamically reallocates cache resources from the cache-rich (backends which don't affect request tail latency) to the cache-poor (backends which affect request tail latency). We evaluate RobinHood with production traces on a 50-server cluster with 20 different backend systems. Surprisingly, we find that RobinHood can directly address tail latency even if working sets are much larger than the cache size. In the presence of load spikes, RobinHood meets a 150ms P99 goal 99.7% of the time, whereas the next best policy meets this goal only 70% of the time.

1 Introduction

Request tail latency matters. Providers of large user-facing web services have long faced the challenge of achieving low request latency. Specifically, companies are interested in maintaining low *tail latency*, such as the 99th percentile (P99) of request latencies [26, 27, 36, 44, 63, 84, 93]. Maintaining low tail latencies in real-world systems is especially difficult when incoming *requests* are complex, consisting of multiple *queries* [4, 26, 36, 91], as is common in *multitier architectures*. Figure 1 shows an example of a multitier architecture: each user request is received by an application server, which then sends queries to the necessary *backends*, waits until all queries have completed, and then packages the results for delivery back to the user. Many large web services, such as Wikipedia [72], Amazon [27], Facebook [20], Google [26] and Microsoft, use this design pattern.

The queries generated by a single request are independently processed in parallel, and may be spread over many

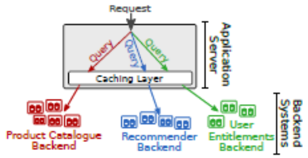


Figure 1: In a multitier system, users submit individual *requests*, which are received by application servers. To complete a request, an application server issues a series of *queries* to various *backend services*. The request is only complete when all of its queries have completed.

backend services. Since each request must wait for all of its queries to complete, the overall *request latency* is defined to be the latency of the request's *slowest* query. Even if almost all backends have low tail latencies, the tail latency of the *maximum* of several queries could be high.

For example, consider a stream of requests where each request queries a single backend 10 times in parallel. Each request's latency is equal to the *maximum* of its ten queries, and could therefore greatly exceed the P99 query latency of the backend. The P99 request latency in this case actually depends on a higher percentile of backend query latency [26]. Unfortunately, as the number of backends in the system increases and the workload becomes more heterogeneous, P99 request latency may depend on different (higher or lower) percentiles of query latency for each backend, and determining what these important percentiles are is difficult.

To illustrate this complexity, this paper focuses on a concrete example of a large multitier architecture: the OneRF page rendering framework at Microsoft. OneRF serves a wide range of content including news (msn.com) and online retail software stores (microsoft.com, xbox.com). It relies on more than 20 backend systems, such as product catalogues, recommender systems, and user entitlement systems (Figure 1).

The source of tail latency is dynamic. It is common in multitier architectures that the particular backend causing high request latencies changes over time. For example,

Thanks!

(And please attend the Scheduling session)