Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models

Suli Yang, *Ant Financial Services Group;* Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *UW-Madison*

https://www.usenix.org/conference/osdi18/presentation/yang

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

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.

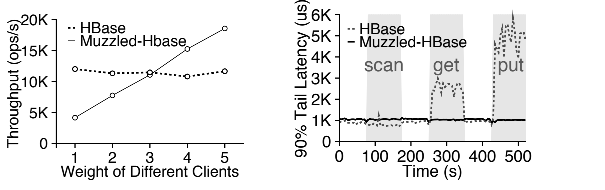
1 Introduction

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 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.



(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.*

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.

consumption patterns and dependencies between components, a TAM effectively links the performance of a storage system to its architecture (while abstracting away implementation details). Using a TAM, various scheduling problems can be discerned, pointing toward solutions that introduce necessary scheduling controls. The system can then be transformed to provide schedulability by fixing these problems, allowing realization of various scheduling policies atop it. TAMs are also readily visualized using *Thread Architecture Diagrams* (TADs), and can be (nearly) automatically obtained by tracing a system of interest under different workloads.

We use TAMs to analyze the schedulability of four important and widely-used scalable storage systems: HBase/HDFS [24, 56], Cassandra [36], MongoDB [15], and Riak [33], and highlight weaknesses in the scheduling architecture of each. Our analysis centers around five essential problems we have discovered, each of which leads to inadequate scheduling controls: a lack of local scheduling control points, unknown resource usage, hidden competition between threads, uncontrolled thread blocking, and ordering constraints upon requests. Fortunately, these problems can be precisely specified using TAMs, enabling straightforward and automatic problem identification. These problems can also be visually identified using TADs, allowing system architects to readily understand where problems arise.

By fixing the problems identified using TAM, HBase, the most complex system we studied, can be transformed to provide schedulability. We show via simulation that Tamed-HBase (TAM-EnableD HBase) utilizes a problem-free thread architecture to enable fair sharing under intense resource competition and provide strong tail latency guarantees with background interference; it also achieves proper isolation despite variances in request amount, size, and other workload factors. We implement Muzzled-HBase (an approximation of Tamed-HBase) to show that TAM-guided schedulability analysis corresponds to the real world.

The rest of this paper is structured as follows. We first introduce the thread architecture model (TAM) (§2), and then discuss how to use TAM to perform schedulability analysis, centered around the five scheduling problems (§3). We use HBase/HDFS as a case study to demonstrate how to use TAM to analyze the schedulability of a realistic system, and make said system schedulable (§4). We then present the schedulability analysis results of other systems (§5). Next, we discuss the limitations of TAM and how it can be extended (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Thread Architecture Model

Implementing new scheduling policies in existing systems is non-trivial; most modern scalable storage systems have complex structures with specific features that complicate the realization of scheduling policies. We introduce *thread architecture models* (TAMs) to describe these structures. The advantage of TAM is that *one can perform schedulability analysis with only information specified in this model, abstracting away all the implementation details.* We first give a general and intuitive description of TAM (§2.1) and describe its visualization using TAD (§2.2). We then discuss how to automatically obtain TAM for existing systems (§2.3). Finally, we give a formal definition of the TAM model (§2.4).

2.1 TAM: General Description

We model scheduling in a storage system as containing requests that flow through the data path consuming various resources while a control plane collects information and determines a scheduling plan to realize the system's overall goal (e.g., fairness). This plan is enforced by local schedulers at different points along the data path.

In modern SEDA-based distributed storage systems, the data path consists of many distinct *stages* residing in different *nodes*. A stage contains threads performing similar tasks (e.g., handling RPC requests or performing I/O). A *thread* refers to any sequential execution (e.g., a kernel thread, a user-level thread, or a virtual process in a virtual machine). Within a stage, threads can be organized as a pool with a fixed (or maximum) number of active threads (*bounded stage*) or can be allocated dynamically as requests increase (*on-demand stage*). In certain stages, some requests may need to be served in a specific order for correctness; this is an *ordering constraint*.

Each bounded stage has an associated queue from which threads pick tasks; each queue is a potential *scheduling point* where schedulers can reorder requests. The queue can be either implicit (e.g., the default FIFO queue of a Java thread pool) or explicit (with an API to allow choice of policy, or hard-coded decisions).

A stage may pass requests to its *downstream* stages for processing. After a thread issues a request to downstream stages, the thread may immediately proceed to another request, or *block* until notified that the request completed at other stages.

Resources are consumed within a stage as requests are processed; we consider CPU, I/O, network and lock[1] resources, but other resources can be readily added to our model. Instead of specifying the exact amount of resources used in each stage (which can change based on specific characteristics of workloads), we only consider whether a resource is *extensively* used in a stage. This simplification allows us to abstract away the details of slightly different workloads but still captures important problems related to resource usage (shown in §3). Exten-

[1]We treat each lock instance as a separate resource, but are usually only interested in one or two highly contended locks in the system, e.g., the namespace lock in HDFS.

C [I] N L name	stage [Boxes above: its resource vector. Stop: ordering constraints]
▬	Scheduling point [Plug: allows pluggable schedulers. No scheduling point: on-demand stage]
A ⟶ B	Downstream relationship [Stage A issues requests to stage B]
A ←-------- B	Blocking relationship [Stage A blocks on the stage B]
▭	Node boundary
C [I] N L	CPU, I/O, network, lock resource [Left to right. Square bracket: unknown usage]

Table 1: **Notation for Thread Architecture Diagrams.**

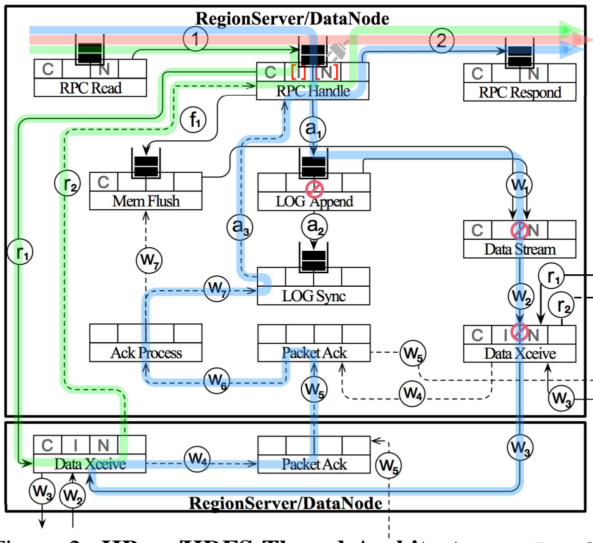


Figure 2: **HBase/HDFS Thread Architecture.** *Based on HBase* 2.0.0 *and Hadoop* 2.7.1. *Some stages are omitted to simplify discussion. Red: main RPC processing flow; green: processing flow that requires HDFS read; blue: processing flow that requires data modifications.*

sive resource usage is interpreted as "any serious usage of resources that is worth considering during scheduling"; we discuss how we choose the threshold in §2.3. If a stage may or may not extensively use a resource during processing based on different workloads, it has *unknown resource usage* for this resource.

All the stages and their collective behaviors, relationships, and resource consumption patterns form the *thread architecture* of a system.

2.2 Visualization with TAD

One advantage of TAM is that it allows direct visualizations using *Thread Architecture Diagrams* (TADs). Table 1 summarizes the building blocks in TADs; Figure 2 through 5 show the TADs of HBase/HDFS [24, 56], MongoDB [15], Cassandra [36] and Riak [33] (labels on the arrows and important workload flows are manually added to aid understanding, and are not parts of TAD). TAM and TAD can be thought of as duals: TAD is the graphical representation of TAM, while TAM is the symbolic representation of TAD; one can easily transform a TAD to its underlying TAM, and vice versa.

We now use the (simplified) HBase/HDFS TAD in Figure 2 to illustrate how to read a TAD and identify spe-

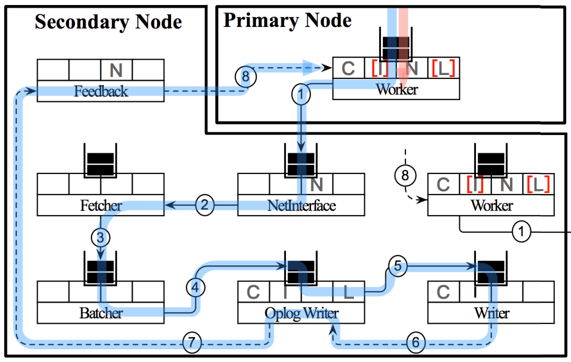


Figure 3: **MongoDB Thread Architecture.** *(v3.2.10) Red: processing flow for requests that do not block on replication; blue: processing flow for requests that do.*

cific features of system scheduling from it.

When HBase clients send queries to the RegionServer, the RPC Read stage reads from the network and passes the request to the RPC Handle stage (1). Based on the request type (Put or Get) and whether data is cached, RPC Handle may have different behavior. One may insert custom schedulers into RPC Handle (plug symbol).

If the RPC needs to read data, RPC Handle checks if the data is local. If not, RPC Handle sends a read request to the Data Xceive stage in a Datanode and blocks ($r_1 - r_2$, where blocking is indicated by dashed r_2). If it is local, RPC Handle directly performs short-circuited reads, consuming I/O. I/O resource usage in RPC Handle is initially unknown and thus marked with a bracket.

For operations that modify data, RPC Handle appends WAL entries to a log (a_1) and blocks until the entry is persisted. LOG Append fetches WAL entries from the queue in the same order they are appended (stop symbol), and writes them to HDFS by passing data to Data Stream (w_1), which sends the data to Data Xceive ($w_2 - w_3$). All WAL entries append to the same HDFS file, so Data Stream and Data Xceive must process them in sequence. LOG Append also sends information about WAL entries to LOG Sync (a_2), which blocks (w_7) until the write path notifies it of completion (further details omitted); it then tells RPC Handle to proceed (dashed a_3). RPC Handle may also flush changes to the MemStore cache (f_1); when the cache is full, the content is written to HDFS with the same steps as with LOG Append writes ($w_1 - w_7$), though without the ordering constraint.

Finally, after RPC Handle finishes an RPC, it passes the result to RPC Respond and continues another RPC (2). In most cases, RPC Respond responds to the client, but if the connection is idle, RPC Handle bypasses RPC Respond and responds directly.

HBase has more than ten complex stages exhibiting different local behaviors (e.g., bounded vs. on-demand), resource usage patterns (e.g., unknown I/O demand), and interconnections (e.g., blocking and competing for the

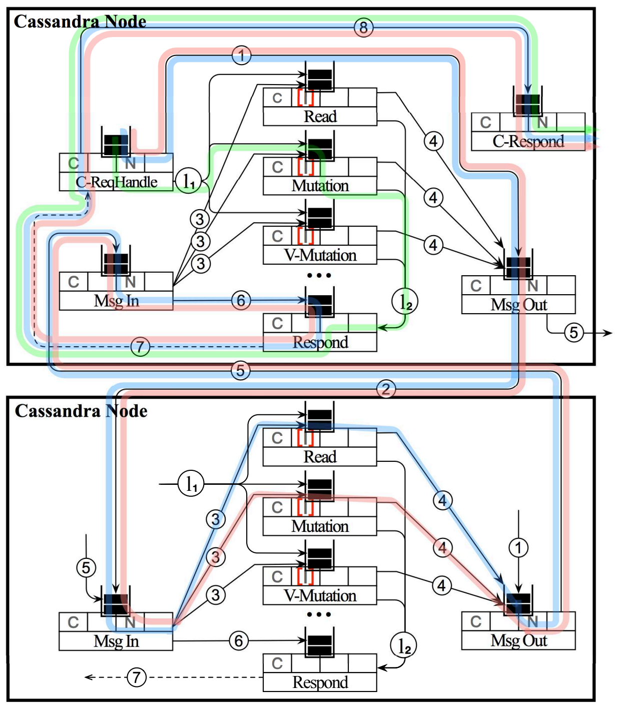


Figure 4: **Cassandra Thread Architecture.** *(v3.0.10) The ellipsis represent other database processing stages. Red: remote mutation processing flow; blue: remote read processing flow; green: local read processing flow.*

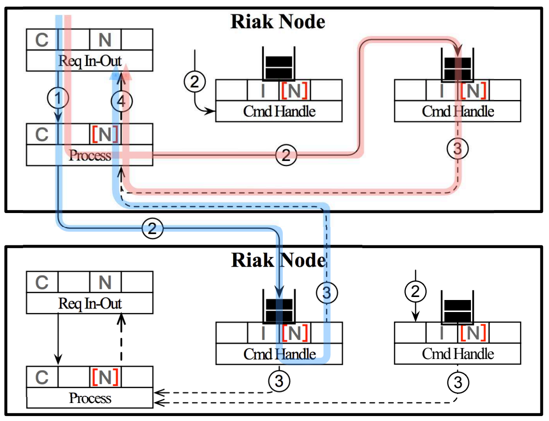
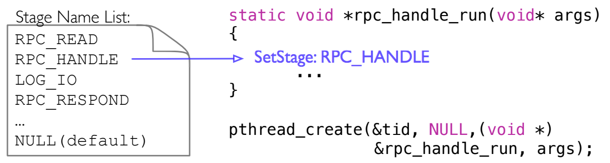


Figure 5: **Riak Thread Architecture.** *(v2.1.4) Red: local request processing flow; blue: remote request processing flow.*



```
Stage Name List:
RPC_READ
RPC_HANDLE
LOG_IO
RPC_RESPOND
…
NULL(default)
```

```
static void *rpc_handle_run(void* args)
{
    SetStage: RPC_HANDLE
        ...
}

pthread_create(&tid, NULL,(void *)
                     &rpc_handle_run, args);
```

Figure 6: **Sample Annotation.**

same resources across stages). All of them are compactly encoded in its TAM/TAD, enabling us to identify problematic scheduling, as we discuss later (§3).

2.3 Automatic Obtainment

TAM is defined with automatic procurement in mind: all information specified in TAM can be (relatively) easily obtained, allowing automation of the schedulability analysis. We now present TADalyzer, a tool we developed to auto-discover TAM/TAD for real systems using instrumentation and tracing techniques. The workflow to generate the TAM of a given system with TADalyzer consists of four steps:

1. *Stage Naming*: the user lists (and names) important stages in the system.
2. *Stage Annotation*: the user identifies thread creation code in the code base and annotates if the new thread belongs to one of the stages previously named. Figure 6 shows a sample annotation. Threads not explicitly annotated default to a special `NULL` stage.
3. *Monitoring*: the user deploys the system and feeds various workloads (e.g., hot-/cold-cached, local/remote) to it. TADalyzer automatically collects necessary information for later TAM generation. If the user missed some important stages in step 1, TADalyzer would notice that some threads in the `NULL` stage are overly active, and alert the user with the stack trace of these threads. Based on the alert, the user identifies the missing stages and repeats step 1-3.
4. *Generating*: After enough information is collected, the user asks TADalyzer to generate the TAM; some information TADalyzer cannot obtain (see Figure 7), and the user needs to provide manually. TADalyzer also automatically plots TAD from the TAM (though the TADs shown in the paper are drawn manually).

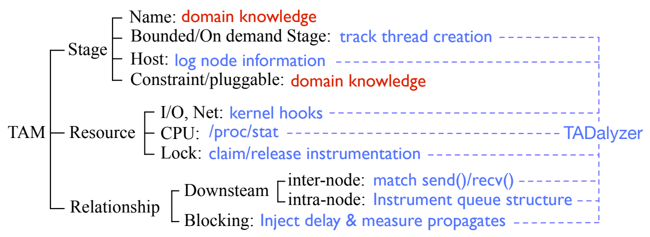
This workflow requires the user to know the important (but not all) stages in the system. From our experience, someone unfamiliar with the code base usually misses naming some stages initially. However, TADalyzer provides enough information to point the user to the code of the missing stages to aid further annotation, and one can typically get a satisfactory TAM within a few (< 5) iterations of the workflow. In HBase and MongoDB such annotation took ~50 lines of code, and ~20 in Cassandra.

We now briefly describe how TADalyzer generates the TAM. Based on user annotation, TADalyzer monitors thread creation and termination, and builds a mapping between threads and stages. Using this mapping, it automatically discovers the following information:

Stage Type: TADalyzer tracks active threads at each stage to classify bounded or on-demand stages.

Resource Consumption: Using Linux kernel tracing tools [1, 40], TADalyzer attaches hooks to relevant kernel functions (e.g., `vfs_read`, `socket_read`) to monitor the I/O and network consumed at each stage. CPU consumption is tracked through */proc/stat*; the lock resource by automatically instrumenting relevant lock operations.

Intra-Node Data Flow: TADalyzer automatically instruments standard classes that are commonly used to



TAM
Stage
Name: domain knowledge
Bounded/On demand Stage: track thread creation
Host: log node information
Constraint/pluggable: domain knowledge
Resource
I/O, Net: kernel hooks
CPU: /proc/stat
Lock: claim/release instrumentation
TADalyzer
Relationship
Downsteam
inter-node: match send()/recv()
intra-node: Instrument queue structure
Blocking: Inject delay & measure propagates

Figure 7: **TADalyzer Summary.** *Black: component in TAM, Blue: how TADalyzer obtain corresponding information in real system, Red: information TADalyzer could not provide.*

pass requests, such as `util.AbstractQueue` in Java and `std<queue>` in C++, to build data flow between stages within the same node.

Inter-Node Data Flow: TADalyzer tracks how much data each thread sends and receives on different ports. By matching the IP and port information, TADalyzer builds the data flow between stages on different nodes.

Blocking Relationship: TADalyzer injects delays in a stage and determines whether other stages block by observing if the delay propagates to these stages.

The current version of TADalyzer cannot automatically derive if a stage provides a pluggable scheduling point or has an ordering constraint, and requests this information from the user. Figure 7 summarizes how TADalyzer obtains TAM information; based on the information, TADalyzer generates the TAM/TAD.

When generating TAM, TADalyzer needs to determine the threshold for extensive resource usage of a stage. In a typical system there exist many "light" stages that are occasionally activated to perform bookkeeping and consume few resources (e.g., a timer thread); even for stages that are actively involved in request processing, they may perform tasks with a particular resource pattern and only very lightly use other resources. When accumulating the resource consumption in a long run, we observe that these stages use at most 1% of the concerned resource, while the stages that are actively consuming resources when processing requests typically use more than 10% (or much higher) of the resource. For example, in MongoDB the Worker stage consumes up to 95% of the total CPU time, while the Fetcher stage consumes at most 0.2%; similarly, in HBase the RPC Respond stage is responsible for 20% to 80% of the total bytes transferred through network, but its I/O consumption never exceeds 1%. TADalyzer thus chooses the extensiveness threshold to be within 1% and 10% to prevent these "light" stages from unnecessarily complicating the TAM (the exact threshold is set to 5%).

TADalyzer has certain limitations (detailed discussion omitted for brevity); in particular, the TAMs generated by TADalyzer are correct, but maybe incomplete (missing stages or flows).[2] However, we would like to emphasize that *TAM defines a clear set of obtainable information* (see §2.4), which *enables* tools that automatically extract this information to construct TAM and perform schedulability analysis. TADalyzer is just one such tool we built to demonstrate the feasibility of automatic schedulability analysis; we encourage other tools to be developed that deploy different techniques (e.g., those in [8, 11, 14, 69]) to discover the information listed in Figure 7 and optimize the process of obtaining TAM.

2.4 TAM: Formal Definition

We now give a more formal definition of the thread architecture model, which precisely specifies the information encoded in a TAM. Such formalism is critical for both automatically constructing TAMs (§2.3) and for systematically identifying the scheduling problems (§3).

Definition 1. A *thread architecture* is defined by the 3-tuple (S, D, B), where

- S is a finite set; each element $s \in S$ is a *stage*, which is defined in Definition 2.
- D is a function that maps S to $\mathbb{P}(S)$ (the power set of S); D represents the *downstream* relationship. For example, $D(s_1) = \{s_2, s_3\}$ means s_1 issues requests to s_2 and s_3.
- B is a function that maps S to $\mathbb{P}(S)$; B represents the *blocking* relationship. For example, $B(s_1) = \{s_2\}$ means stage s_1 blocks on stage s_2.

Definition 2. A *stage* is defined by the 5-tuple (n, h, r, o, q), where

- n is a string representing the name of the stage.
- h is a positive integer indicating host ID. Stages with the same h value are on the same node.
- r is a 4-vector representing the resource usage pattern of this stage. Each component in r can take one of the three values: `true`, `false`, or `unknown`, indicating whether the corresponding resource (CPU, I/O, network, lock) is used *extensively* in this stage.
- o is a boolean value representing whether the stage has an ordering constraint or not.
- q represents the local scheduling type, and can take one of the three values: `on_demand`, `pluggable` or `general`, indicating whether the stage is on-demand, allows pluggable schedulers, or has hard-coded or implicit scheduling logic.

3 Scheduling Problems

TAM allows us to identify scheduling problems without being concerned about low-level implementation details; it also points towards solutions that introduce necessary scheduling controls. We now discuss how to perform schedulability analysis using TAM/TAD. Our analysis

[2]All TAM/TADs shown in the paper (except MongoDB) have been validated by each system's developers [21, 23, 27].

centers around five common problems we discovered in modern distributed storage systems: *no scheduling*, *unknown resource usage*, *hidden contention*, *blocking*, and *ordering constraint*. To illustrate the process clearly, we begin by focusing on systems with only a single problem; in Section 4 we consider the HBase TAM in which multiple problematic stages are interconnected.

For each problem, we first give a general description, then precisely specify the problem in TAM and TAD. We use simulation to demonstrate how problematic thread architecture hinders scheduling policy realization; different scheduling policies including fairness, latency guarantees, and priority scheduling are investigated.

The simulation framework (built on simpy [39]) provides building blocks such as requests, threads, stages, resources, and schedulers. Using TAMs as blueprints, stages can be assembled to form various thread architectures that refect existing or hypothetical system designs. With a given thread architecture, one can specify workload characteristics (e.g., request types and arrival distribution), resource configurations (e.g., CPU frequency and network bandwidth), and scheduling policies; the framework then simulates how requests flow through the stages and consume the resources, and reports detailed performance statistics.

Unless noted, all simulations in this section use a common configuration: two competing clients (C1 and C2) continuously issue requests; C1 has 40 threads, C2 varies; each node has a 1 GHz CPU, 100 MB/s disk, and a 1 Gbps network connection.

3.1 No Scheduling

Each resource-intensive stage in a thread architecture should provide local scheduling. With local scheduling for a stage, requests are explicitly queued and resource-intensive activities can be ordered according to system's overall scheduling goal. In contrast, an on-demand stage with no request queue and extensive resource usage suffers the *no scheduling* problem (e.g., the Data Stream and Data Xceive stages in HBase, and the Req In-Out and Process stages in Riak).

TAM: A TAM (S, D, B) suffers no scheduling if $\exists s \in S$, s.t. $s.r \neq$ `[false, false, false, false]` $\wedge\ s.q =$ `on_demand`. [3]

TAD: A TAD suffers no scheduling if it contains stages with non-empty resource boxes but no queues.

Figure 8(a) shows a simple TAD with two stages, the second of which has no scheduling (an on-demand stage with intensive I/O). The scheduler for Req Handle (Q1) attempts to provide a latency guarantee to C1 using earliest-deadline-first (EDF) but is unsuccessful: as

[3]A more stringent definition may require each resource intensive stage to provide `pluggable` scheduling point to allow flexible scheduling policy realization; we opt for a looser definition here.

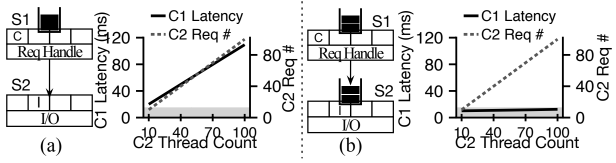


Figure 8: **The No Scheduling Problem.** *Each client request requires 100 us CPU and 100 KB I/O, making I/O the bottleneck. The deadline is set to 15 ms for C1, 500 ms for C2; the gray area indicates latency within the C1 deadline. The left y-axis shows the average latency of C1; the right y-axis shows the number of C2 requests competing with C1 at the I/O stage.*

C2 issues requests with more threads, the latency of C1 exceeds the deadline by as much as 5x. The problem occurs because Q1 scheduling is irrelevant when Req Handle is not the bottleneck: the average queue length of Q1 is zero. Meanwhile, as shown in Figure 8(a), there are many requests contending for I/O in the I/O stage, which is not managed by a scheduler.

Figure 8(b) shows another architecture which has the same functionality but does not suffer the no scheduling problem as it possesses a scheduling point at the I/O stage. Local scheduling points enable the system to regulate I/O resource usage at the point where the resource is contended, thus simply and naturally ensuring latency guarantees and isolation of the two clients.

3.2 Unknown Resource Usage

Each stage within a system should know its resource usage patterns. However, in some stages, requests may follow different execution paths with different resource usage, and these paths are not known until after the stage begins. For example, a thread could first check if a request is in cache, and if not, perform I/O; the requests in this stage have two execution paths with distinct resource patterns and the scheduler does not know this ahead of time. In such cases, the stage suffers *unknown resource usage* (e.g., the RPC Handle stage in HBase due to the short-circuited reads it might perform). Unknown resource usage forces schedulers to make decisions before information is available.

TAM: A TAM (S, D, B) suffers unknown resource usage if $\exists s \in S$, $\exists i \in \{1, 2, 3, 4\}$, s.t. $s.r[i]$ = `unknown`.

TAD: A TAD suffers unknown resource usage if it contains resource symbols surrounded by square brackets.

Figure 9 (a) shows a single stage with unknown I/O usage (the bracket around the I/O resource), where Q1 performs dominate resource fairness (DRF) [26] with equal weighting. When C2 issues a mix of cold and cached requests, Q1 schedules C2-cold and C2-cached in the same way. Even though there are idle CPU resources, Q1 cannot schedule additional C2-cached requests to utilize the CPU because it does not know whether the request would later cause I/O, which is currently contended. Unknown resource usage thus causes low CPU utilization and low throughput of C2-cached.

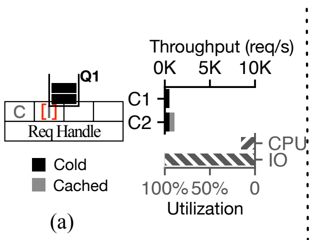
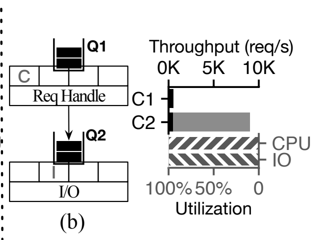



Figure 9: **The Unknown Resource Usage Problem.** *Both C1 and C2 issue requests that require 100 us CPU time and 100 KB I/O. C2 also issues cached requests that requires only 100 us CPU but no I/O. In (a) the Req Handle threads first look up the cache when serving a request, and perform I/O if it is a cache miss. In (b) a separate I/O stage performs I/O.*

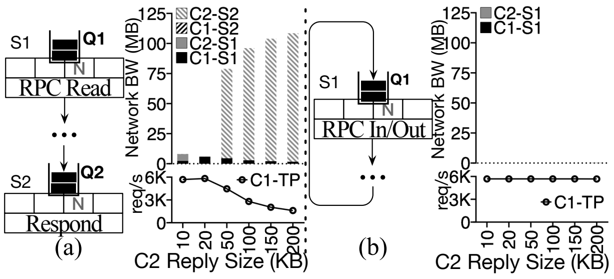


Figure 10: **The Hidden Contention Problem.** *C1 sends 1 KB requests and receives 10 KB replies; C2 also sends 1 KB requests but its reply size varies (shown in the x-axis). The line graph below shows the throughput of C1. The bar graph above shows the bandwidth each stage is forced to allocate to C1 or C2 to maintain work conservation: when scheduling, there are only C1/C2 requests in the queue. C1-S1 means the bandwidth S1 (Req Read) is forced to allocate to C1, and so on.*

Figure 9(b) shows another system with the same functionality but one stage split into two. The Req Handle stage performs CPU-intensive cache lookups while a new stage performs I/O for requests that miss the cache. Each stage has its own scheduler. Q1 freely admits requests when there are enough CPU resources, leading to high CPU utilization and C2-Cached throughput. Meanwhile, not only does Q2 know a request needs I/O, it also knows the size and location of the I/O, enabling Q2 to make better scheduling decisions. System(b) is thus free from the unknown resource usage problem.

3.3 Hidden Contention

When multiple stages with independent schedulers compete for the same resource, they suffer from *hidden contention* which impacts overall resource allocation (e.g., the Worker and Oplog Writer stages in MongoDB for database locks, and the Read, Mutation, View-Mutation stages in Cassandra for CPU and I/O). The hidden contention in MongoDB is reported to cause unbounded read latencies in production [6]. Hidden contention is ubiquitous, because some contention is difficult to avoid (e.g., most stages use CPU).

TAM: A TAM (S, D, B) suffers hidden contention if $\exists s_1 \in S$, $\exists s_2 \in S$, $\exists i \in \{1, 2, 3, 4\}$ s.t. $s_1 \neq s_2 \wedge s_1.h = s_2.h \wedge s_1.q \neq \texttt{on\_demand} \wedge s_2.q \neq \texttt{on\_demand} \wedge s_1.r[i] \neq \texttt{false} \wedge s_2.r[i] \neq \texttt{false}$.

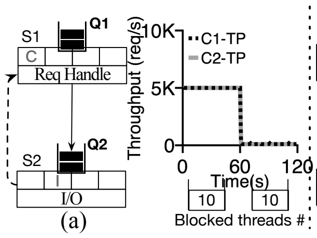
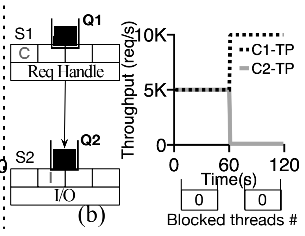



Figure 11: **The Blocking Problem.** *Initially both C1 and C2 issue requests that require 100 us CPU time and can be completed within the Req Handle Stage. At time 60, C2 switches to an I/O intensive workload where each request additionally requires 100 KB I/O at the I/O stage. C1 continues to issue CPU-only requests. The table below shows the average number of blocking threads in Req Handle (10 threads in total).*

TAD: A TAD suffers hidden contention if it contains stages within a node boundary that have separate queues but the same resource in the resource usage boxes.

Figure 10(a) shows a two-stage system with the network as the source of hidden contention; one stage reads requests and the other sends replies. Both Q1 and Q2 preform fair queuing [26] with equal weighting. However, enforcing fairness at each stage does not guarantee fair sharing at the node level. When C2 increases its reply size (i.e., its network usage), it unfairly consumes up to 95% of the network and reduces throughput of C1. With larger C2 reply size, S2 is frequently forced to schedule C2 because there are no requests from C1 in its queue. As there is no regulation on contention between stages, S2 effectively monopolizes the network when it sends larger replies (on behalf of C2) and prevents S1 from using the network; this causes fewer requests to be completed at S1 and flow to S2, further limiting the chocies available to S2. Hidden network contention between the two stages thus causes unfair scheduling.

Figure 10(b) shows a system where one stage handles both reading and replying RPCs. Q1 has full control of the network and can isolate C1 and C2 perfectly.

3.4 Blocking

For optimal performance, even when some requests are waiting to be serviced, each stage should allow other requests to make progress if possible; a problem occurs if there are no unblocked threads to serve these requests. A system has a *blocking* problem if a bounded stage may block on a downstream stage (e.g., the RPC Handle stage in HBase, and the Worker stage in MongoDB), as scenarios may occur where all threads in that stage block at one path and other requests that could have been completed cannot be scheduled. The blocking problem of HBase is reported to cause extremely low throughput or even data loss in production [5]. Blocking forces upstream schedulers to account for downstream progress.

TAM: A TAM (S, D, B) suffers blocking if $\exists s \in S$, s.t. $s.q \neq \texttt{on\_demand} \wedge B(s) \neq \emptyset$.

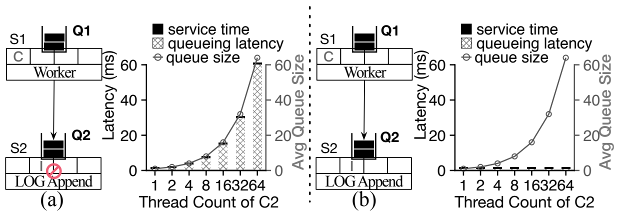


Figure 12: **The Ordering Constraint Problem.** *High priority C1 issues requests in burst; low priority C2 steadily issues requests with more threads. Each request requires 100 us CPU time at the Worker stage, and 100 KB I/O at the LOG Append stage. The left y-axis shows the average latency of C1; the right y axis shows the average queue size of LOG Append.*

TAD: A TAD suffers blocking if it contains stage boxes with queues and dashed arrows pointing to them.

Figure 11(a) shows a system with blocking at Req Handle. Requests in Req Handle have two paths: they may complete in this stage or block on the I/O stage; the schedulers perform DRF [26] with equal weighting. Initially both C1 and C2 receive high throughput as they issue cached requests without blocking; however, when C2 switches to an I/O-intensive workload, the throughput of C1 (which is still CPU-only) suffers. The table below shows that all threads in Req Handle are blocked on I/O, leaving no threads to process C1 requests.

In contrast, Figure 11(b) shows a system in which the Req Handle stage is asynchronous. No threads block; all perform useful work, leading to high throughput for C1.

3.5 Ordering Constraint

Many storage systems use Write-Ahead Logging (WAL), which requires the writes to the log to occur in sequence. When a system requires some requests at a resource-intensive stage to be served in a specific order to ensure correctness, it has the *ordering constraint* problem (e.g., the Data Stream and Data Xceive stage in HBase). Ordering constraint leaves the scheduling framework with fewer or no choices, because the local scheduler cannot reorder resource-intensive activities as desired.

TAM: A TAM (S, D, B) suffers ordering constraint if $\exists s \in S$, $\exists i \in \{1,2,3,4\}$ s.t. $s.o = \texttt{true} \wedge s.r[i] \neq \texttt{false}$.

TAD: A TAD suffers ordering constraint if it contains stages with stop symbols and non-empty resource boxes.

Figure 12(a) shows a two-stage system with ordering constraint on the second stage. The schedulers enforce priorities, where high priority requests are served first as long as this does not break correctness. In this system, C1 (high priority) suffers much longer latency when C2 (low priority) issues requests aggressively. The majority of this latency occurs from queuing delay in the second stage since low priority requests must be serviced first if they enter the stage earlier.

Figure 12(b) shows a system that eliminates the problem by separating requests from different clients into different streams that share no common states (e.g., each stream has its own WAL); even though requests within a stream are still serviced in order, the scheduler can choose which stream to serve and provide differentiated services on a per-stream basis. The figure shows that C1 maintains low latency despite the larger queue size at the LOG Append stage when C2 issues more requests: free from the ordering constraint, Q2 can pick the high priority requests from C1 first.

3.6 Discussion

We have identified five categories of scheduling problems. For each category, we have given an example that highlights the problem. In some cases the example highlights a fairness problem; in others it highlights a latency or utilization problem. However, one should note that each of these problems can manifest in many different ways, causing violations in any scheduling discipline. For example, in §3.1 we show how no scheduling causes excessive latencies; since there are no scheduling points to prioritize requests, it could as easily cause unfairness or priority inversions. How (and whether) the scheduling problems manifest depends on the resources available, the workload, and the scheduling policy; when TAM/TAD suggests a scheduling problem, it means that there exist certain workloads/resource configurations under which the problem manifests.

Each of the five scheduling problem by itself is not very surprising. However, by compactly representing the thread architecture and exposing scheduling problems, TAM can serve a useful conceptual tool that allows the system designer to identify and fix *all* the problems in an existing system, or to design a problem-free architecture for a new system. In addition, TAD enables visual analysis, making it clear where problems arise, while the TAM-based simulation can be used to study how scheduling problems actually manifest given certain workloads and resource configurations.

Do the five categories of problems exhaustively describe how system structure could hinder scheduling? For now we can only answer this question empirically. We analyzed systems with distinct architectures (thread-based vs. loose SEDA vs. strict SEDA) and thread behaviors (kernel- vs. user-level threads). Only these problems arise and fixing them allows us to realize various scheduling policies effectively. We leave proving the completeness of the problems to future work.

4 HBase: A Case Study

Given the TAM of a system, multiple scheduling problems may be discovered, pointing towards solutions that introduce necessary scheduling controls. By fixing these problems, the system can be transformed to provide schedulability. We now perform such analysis on a realistic storage system, the HBase/HDFS storage stack (hereinafter just HBase). We focus on

HBase, as it presents the most complex architecture, is widely deployed in many production environments [24], and achieving schedulability remains difficult despite repeated attempts [3, 5, 37, 63, 68]. We analyze the schedulability of MongoDB [15], Cassandra [36] and Riak [33] later (§5).

4.1 TAM simulations

We simulate an HBase cluster with 8 nodes; one master node hosts the HMaster and NameNode, and 7 slave nodes host RegionServers and DataNodes. Each node has a 1 GHz CPU, 100 MB/s disk, and a 1 Gbps network connection. Using this simulation, we compare the orignal HBase (Orig-HBase) and the HBase modified with our solutions (Tamed-HBase, with its TAD shown in Figure 14); later we implement the solutions to show that our TAM-based simulation corresponds to the real world. The solutions can be used to realize any scheduling policy; in our simulation the schedulers simply attempt to isolate C1's performance from C2's workload changes.

4.1.1 No Scheduling

Problem: The Data Xceive and Data Stream stages in HBase have a non-empty resource vector and `on_demand` scheduling type, indicating the no scheduling problem.
Solution: In the Tamed-HBase TAM, we change the scheduling type of Data Xceive and Data Stream from `on_demand` to `pluggable`, so it is free from no scheduling. In a real system, this corresponds to adding scheduling points to the two stages and exporting an API to allow different schedulers to be plugged into each.

We simulate a workload where C1 and C2 keep issuing (uncached) Gets, each of which incurs 128 K I/O at Data Xceive. C1 has 40 threads issuing requests in parallel; the number of threads of C2 increases from 40 to 200. Figure 13(a) shows that even though the original TAM does not isolate C1 from C2, our modified TAM provides stable throughput to C1 despite the change of C2.

4.1.2 Unknown Resource Usage

Problem: In HBase TAM, the I/O and network components of the RPC Handle resource vector take the `unknown` value, indicating unknown resource usage.

Further code inspection reveals that the RPC Handle threads only sends responses when the network is idle, so it does not interfere with scheduling. TAM produces a false positive here because the threads exhibited "scheduler-like" behavior (deciding whether to perform a task based on the status of the resource) without going through the schedulers, which is not captured by TAM. Short-circuited reads, which are unknown when the request is scheduled, do cause contention for I/O and lead to ineffective scheduling.
Solution: We remove the unknown resource usage in the RPC Handle stage by moving short-circuited reads from RPC Handle to Data Xceive. Instead of performing reads by itself, once the RPC Handle stage recognizes a short-circuited read, it directly passes the read to the local Data Xceive stage without going through network transferring; at this point, the Data Xceive scheduler has knowledge of the I/O size and locations.

We simulate a standalone HBase node, which ensures that all HDFS reads at the RegionServer are short-circuited, thus isolating the effect of unknown resource usage. In Figure 13(b), both C1 and C2 issue Gets on cold data, which incurs 100 KB short-circuited reads at RPC Handle. C2 also issues cached Gets that do not require I/O. One can see that Tamed-HBase achieves additional throughput for the cached Gets of C2 compared to Orig-HBase, without reducing the throughput of C1 or C2's cold-cached Gets.

4.1.3 Hidden Contention

Problem: Within the same node of the HBase TAM, both the RPC Handle and Data Xceive stages have an I/O component in their resource vectors; the RPC Read, RPC Handle, RPC Respond, Data Stream, and Data Xceive stage resource vectors all share the network component; many stage resource vectors contain the CPU component. All of them lead to the hidden contention problem.
Solution: To remove hidden contention, we restructure the stages so that in Tamed-HBase, each resource is managed by one dedicated stage. In general, one cannot completely eliminate hidden contention by dividing stages based on resource usage for two reasons:

1. Without special hardware, network packet processing requires significant CPU [16, 20, 30], so the network stage inevitably incurs both network and CPU usage.
2. Lock usage typically cannot be separated to a dedicated stage: it may be pointless to obtain a lock without doing some processing and consuming other resources.

In the case of HBase, the highly contended namespace lock is obtained to perform namespace manipulation (not shown in the simplified TAD), which does not incur extensive usage on other resources, so the lock stage can be separated. The network stage in Tamed-HBase does incur CPU usage; however, by moving most CPU intensive tasks to the CPU stage (e.g., serialization and checksum verification), we can reduce the hidden contention on CPU between the network stage and the CPU stage to a minimal level. The restructured stages are shown in Figure 14; to avoid no scheduling, all the new stages have `pluggable` scheduling points, but the blocking relationships and order constraints are inherited from the old stages to the new ones (until further fixes).

We simulate a workload where C1 and C2 keep issuing 1 KB RPC requests. C1's response size remains 20 KB, while C2's response size varies from 10 to 200 KB. Figure 13(c) shows Tamed-HBase, with the hidden con-

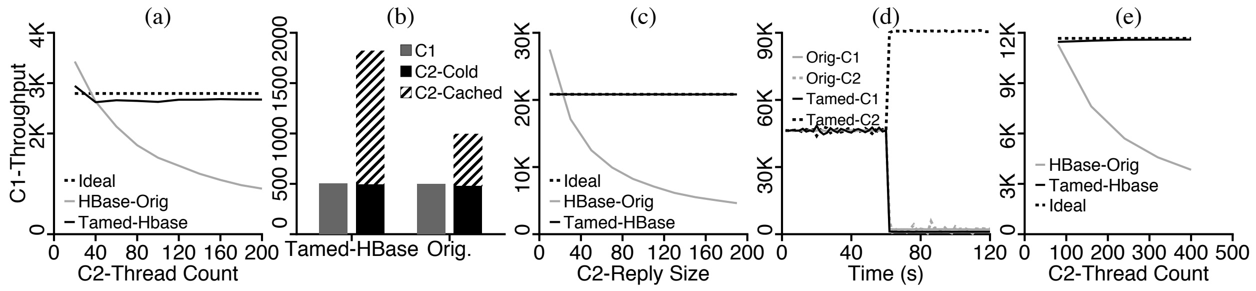


Figure 13: **Tamed-HBase Simulation.** *Problem and solution for (a) no scheduling; (b) unknown resource usage; (c) hidden contention; (d) blocking; (e) ordering constraints.*

tention on network removed, isolates C1 from C2's reply size change; Orig-HBase cannot provide isolation.

4.1.4 Blocking

Problem: In the HBase TAM, three stages are bounded and have a non-empty blocking set: RPC Handle, Mem Flush, and Log Sync, suggesting the blocking problem (which actually occurs in production [5]).

Solution: In Tamed-HBase (with stages restructured to remove hidden contention), we make the CPU and Log Sync stage asynchronous to fix the blocking problem.

In Figure 13(d) we simulate a workload where initially both C1 and C2 issue cached Gets. At time 60s C2 request uncached data, causing threads to block on I/O. When C2 switches to an I/O intensive workload, Tamed-HBase allows C1 to achieve high throughput. In contrast, Orig-HBase delivers very low throughput even though the system has enough resources to process C1 requests.

4.1.5 Ordering Constraints

Problem: In the HBase TAM, the Data Stream and Data Xceive stage have ordering constraints and resource usages, which points to the ordering constraint problem (the Log Sync stage also has ordering constraint, but does not incur extensive usage on any resources, so does not lead to the ordering constraint problem).

Solution: By re-designing the consistency mechanism, the ordering constraint can be removed. For example, each client can maintain a separate WAL, thus eliminating the need to preserve request ordering across clients and removing the ordering constraint in the Log Append and I/O stage in Tamed-HBase.

We simulate a workload where both C1 and C2 issue 1 KB Puts, resulting in 1 KB WAL appends. Figure 13(e) shows that unlike in Orig-HBase, where the throughput drops sharply as C2 issues more requests, Tamed-HBase, with the ordering constraint removed, is able to isolate C2's effect on C1.

4.1.6 Discussion

HBase does attempt to provide scheduling, in the form of exporting a scheduling API at the RPC Handle stage; however, this effort is rather incomplete as it fails to solve any of the scheduling problems HBase possesses, thus suggesting the importance of systematic schedulability analysis. The TAD of Tamed-HBase is shown in Figure 14. With the aid of TAM, we are able to identify and solve all of HBase's scheduling problems (except for the hidden contention on CPU, which we reduce to a low level), and transform HBase to provide schedulability.

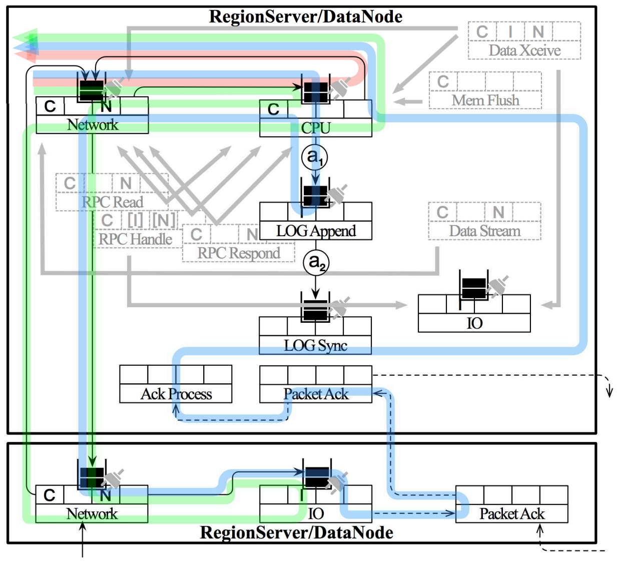


Figure 14: **Tamed-HBase Thread Architecture.** *Stages in grey are replaced by new stages.*

4.2 Implementing Schedulable HBase

In this section, we demonstrate that real HBase suffers from the scheduling problems we identified, and fixing these problems leads to schedulability. The schedulable HBase implementation gives us experience realizing schedulability in real systems and validates that the TAM simulations are excellent predictors of the real world.

To match the simulation environment, we run experiments on an 8-node cluster. Each node has two 8-core CPUs at 2.40 GHz (plus hyper-threading), 128 GB of RAM, an 480 GB SSD (to run the system) and two 1.2 TB HDD (to host the HDFS data). The nodes are connected via 10 Gbps network. One node hosts the HMaster, NameNode, and Secondary NameNode; the other seven nodes host RegionServers and DataNodes.

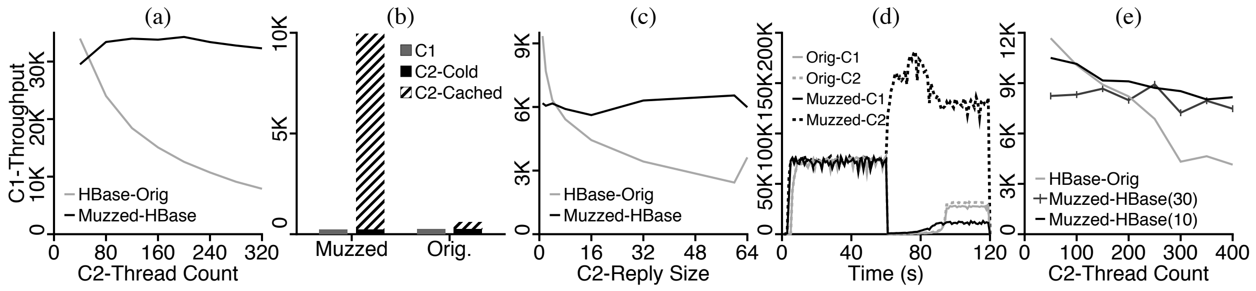


Figure 15: **Muzzled-HBase Implementation.** *The sub-figures repeat Figure 13(a)-(e), respectively (the axis scales are different). To isolate the effect of short-circuit reads, in (b) a standalone node is used instead of a cluster. To ensure that the RegionServer network bandwidth is the bottleneck, in (c) the bandwidth is limited to 100 Mpbs using dummynet [49].*

4.2.1 Schedulability in Real Implementation

In §4.1 we showed via simulation how Tamed-HBase solves the scheduling problems by changing the thread architecture. However, realizing these changes in implementation may be too difficult and risky; for example, making stages asynchronous requires changing the RPC programming model of HBase, and removing the ordering constraint is akin to re-designing the consistency mechanism. Thus in implementation we use various approaches to alleviate the effects of the scheduling problems and approximate the control achieved by Tamed-HBase, with minimal changes to the TAM; we call the resulted implementation Muzzled-HBase (for it is not completely tamed).

No Scheduling: We add scheduling points to the Data Xceive and Data Stream stages to fix the no scheduling problem in HBase. Figure 15(a) illustrates that HBase suffered the no scheduling problem and as a result, the throughput of client C1 is significantly harmed when C2 issues more requests; further, it shows that adding scheduling points at resource-intensive stages provides performance isolation in the real-world.

Unknown Resource Usage: For ease of implementation, in Muzzled-HBase we do not move short-circuited read processing to Data Xceive, as we did for Tamed-HBase (§4.1.2). Instead, we keep the TAM unchanged and use speculative execution to work around this problem. We track the workload pattern of each client; when the CPU is idle, we speculatively execute requests from the CPU-intensive clients. If, during speculation, a request is found to require I/O, it is aborted and put back on the queue where it is subjected to normal scheduling.

The unknown resource problem that exists in HBase is shown in Figure 15(b): when client C2 requests in-cache data, Orig-HBase is not able to efficiently utilize the CPU. Muzzled-HBase with speculative execution dramatically improves the throughput of C2 without harming C1, achieving roughly the same effects as Tamed-HBase, though at the cost of wasted CPU cycles.

Hidden Contention: The complete solution to the hidden contention problem requires restructuring the TAM; this is further complicated by the fact that these stages reside in two separate processes (RegionServer and DataNode). For implementation simplicity, in Muzzled-HBase we only combine the RPC Read and RPC Respond stage, which are mostly responsible for network resource consumption. We work around the remaining contention by having a controller monitor resource usage and adjust client weights at each stage. If stage S1 is excessively using resource on behalf of client C1, the weight of C1 is reduced across *all* stages so that fewer C1 requests are issued to S1, forcing S1 to either use fewer resources or serve other clients; this algorithm is similar to the one deployed in Retro [37].

Figure 15(c) verifies that HBase suffered hidden contention across multiple stages, which manifests when one stage consumes more resources on behalf of a particular client (i.e., more network for C2). The small difference between the implementation and simulation results for a reply size of 64KB occurs because in the implementation, after transferring 64KB, the RPC Respond thread switches to another request; we did not simulate this detail. With two network-intensive stages combined and cross-stage coordination, Muzzled-HBase is able to control the hidden contention and largely ensures isolation, though incurs extra communication overheads.

Blocking: We work around the blocking problem in the RPC Handle stage in HBase without changing the RPC programming model by treating RPC Handle threads as a resource and allocating them between clients like CPU or I/O resources. This approach does not eliminate blocking, but prevents one client from occupying all threads and allows other clients to make progress.

The blocking problem that exists within HBase is illustrated in Figure 15(d). In Orig-HBase, when the workload of one client switches from CPU to I/O-intensive (C2 at time 60), both clients are harmed because not enough threads are available. Our solution, however, protects C1 from the workload change of C2. The slight difference in the implementation and simulation results occurs because we did not simulate page cache effects.

Ordering Constraint: Directly removing ordering constraints from the TAM would require re-designing the consistency mechanism of HBase. In Muzzled-HBase,

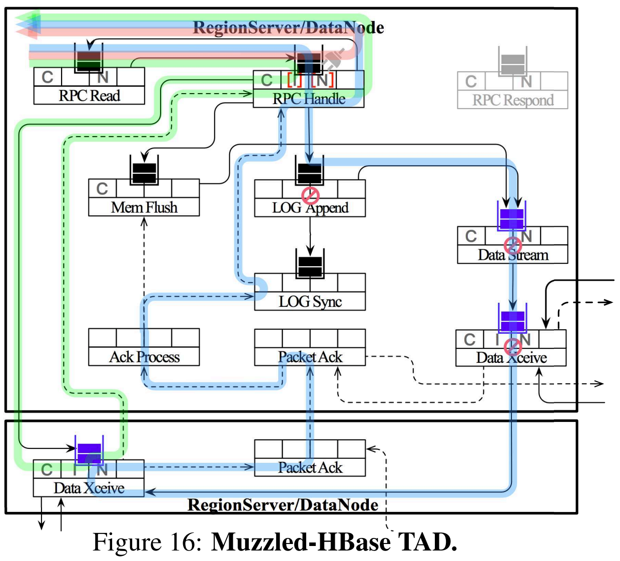


Figure 16: **Muzzled-HBase TAD.**

we work around this by scheduling at RPC Handle, above the ordering-constrained LOG Append stage. Note that we already schedule based on RPC Handle time in this stage to solve the blocking problem. Since threads block until WAL writes are done, under a stable workload, blocking time is roughly proportional to the number of downstream requests, and scheduling RPC Handle time indirectly schedules the WAL writes before passing it to LOG Append. However, the number of RPC Handle threads are typically larger than the I/O parallelism in the system, making this approach less effective; therefore, we compare two settings of Muzzled-HBase, with 10 or 30 RPC Handle threads.

HBase's ordering problem is shown in Figure 15(e); when C2 writes more data, the throughput of C1 suffers. Again, this problem is alleviated in Muzzled-HBase by limiting the number of outstanding requests to the lower stage to 10 or 30; 30 outstanding requests leads to worse isolation than 10, as C1 competes with more requests from C2 after they enter RPC Handle.

Summary: The TAD of Muzzled-HBase is shown in Figure 16. We can see that the no scheduling problem and the hidden contention between RPC Read and RPC Respond are fixed. However, it still exhibits other problems, including unknown resource usage, blocking, ordering constraint, and hidden contention among other stages; changing the thread architecture of HBase to fix these problems would be too difficult. Various approaches are used instead to mitigate the effects of these problems and achieve approximate scheduling control, but these approaches also incur overheads (e.g., wasted CPU cycles on aborted requests or communication across stages).

On top of Muzzled-HBase, multiple scheduling policies are implemented, including FIFO, DRF and priority scheduling. Client identifiers are propagated across stages with requests, so each scheduler can map requests back to the originating client. In our implementation, a centralized controller collects information and coordinates local scheduler behavior; however, other mechanisms such as distributed coordination are also possible. For now the scheduler only performs network resource scheduling on server bandwidth; we anticipate incorporating global network bandwidth allocation [35] in the future. The final implementation consists of ∼4800 lines of code modification to HBase and ∼3000 to HDFS.

The performance of Muzzled-HBase for YCSB [17] is shown in Figure 1. For Figure 1(a), five clients are each given a different weight and we use DRF-based local scheduler to achieve global weighted fairness. Orig-HBase was unable to provide weighted fairness across clients with different priorities, instead delivering approximately equal throughput to each; Muzzled-HBase, in contrast, enforces weighted fairness as desired. For Figure 1(b), priority scheduling is implemented atop Muzzled-HBase by always reserving a small subset of resources, including the RPC Handle threads for foreground workloads. With Orig-HBase, the tail latencies of the foreground workload increase significantly when different types of workloads run in background; Muzzled-HBase, however, is able to maintain stable latencies despite the interference from the background workloads.

4.2.2 Discussion

Schedulability can be achieved by modifying the problematic TAM to eliminate scheduling problems. However, as we can see in the case of HBase, changing the TAM for existing systems usually involves restructuring the system, which is labor-intensive. To minimize the changes to the architecture or lower the engineering effort, often we are forced to keep the same TAM, but use various approaches to work around its inherent structural flaws and alleviate the effects of the scheduling problems. Unfortunately, these approaches only provide approximate scheduling control and incur overheads.

We thus encourage developers to take schedulability into consideration in the early phase of system design; this is especially important in a cloud-based world where users demand isolation and quality of service guarantees. By specifying the TAM of a system, potential scheduling problems can be discovered early, avoiding the painful process of retrofitting scheduling control later. Of course, schedulability may need to be balanced with other system design goals. For example, the system architects may decide that having a simple synchronous programming model is more important, and accept blocking at some stages. However, these kind of compromises should be made only after carefully weighing the trade-offs between different goals, not just due to the obliviousness of their schedulability ramification.

5 Schedulability of Other Systems

Earlier we showed how to transform HBase to provide schedulability. Other concurrent systems can be analyzed and tranformed in the same way. Here, we analyze the schedulability of MongoDB [15], Cassandra [36], and Riak [33]. Table 2 presents a summary of their scheduling problems. Some of the problems predicted by TAM have been experienced in production environments [4, 5, 6]; these problems and their solutions have also been verified by simulation results (see [67]).

MongoDB: The TAD of MongoDB is shown in Figure 3. From its TAM we can identify (a) the unknown resource usage problem at the Worker stage, which processes client requests until completion; (b) the hidden contention problem in the secondary node; most notably, the Worker and Oplog Writer stages compete for database locks, causing reads to have unbounded delay under heavy write load, which is reported in production [6]; (c) the blocking problem at the Worker stage.
Lessons: MongoDB resembles the traditional thread-per-request architecture and thus suffers unknown resource usage, which stems from the complex execution path within one thread. The complex path and resource patterns within the Worker stage makes it challenging to work around this problem. We expect that altering MongoDB to provide schedulability will be difficult and may require substantial structural changes.

Cassandra: The TAD of Cassandra is shown in Figure 4. From its TAM we identify (a) unknown resource usage in the Read, Mutation, View-Mutation stages since those stages may perform I/O; (b) hidden contention between many stages for CPU, I/O and network; (c) blocking in the C-ReqHandle stage.
Lessons: Cassandra closely follows the standard SEDA architecture, where all activities are managed in controlled stages; unfortunately, schedulability does not automatically follow. Too many stages with the same resource pattern leads to hidden contention and the "inability to balance reads/writes/compaction/flushing", as reported by developers [4]; likewise, CPU- and I/O-intensive operations in the same stage leads to unknown resource usage. More thoughts on how to divide stages are needed to build a highly schedulable system. Instead of dividing stages based on functionality, we recommend dividing stages based on resource usage patterns to give more resource information to the scheduler and reduce hidden competition. Cassandra is currently moving toward this direction: developers have proposed combining different processing stages into a single non-blocking stage, and moving I/O to a dedicated thread pool [4].

Riak: The TAD of Riak is shown in Figure 5. From its TAM we can identify (a) the no scheduling problem at the Req In-Out and Req Process stages; (b) unknown resource usage in the Process and Cmd Handle stages; (c) hidden contention across all stages.
Lessons: Riak also closely follows the SEDA architecture. Riak relies heavily on light-weighted processes and transparent IPC provided by the Erlang virtual machine, which makes resource management implicit [22]. Creating a new Erlang process may have low overhead; creating them on-demand leads to the no scheduling problem. Similarly, with transparent IPC, many stages may consume network bandwidth without knowing it, causing unknown resource usage and hidden contention. To make Riak schedulable, one must either explicitly manage the above mechanisms, or change Erlang VM to allow scheduling policies to be passed from Riak to the VM, which manges the resources.

	N	U	C	B	O
HBase [24]	✖	✖	✖	✖	✖
MongoDB [15]		✖	✖	✖	
Cassandra [36]		✖	✖	✖	
Riak [33]	✖	✖	✖		

Table 2: **Scheduling Problems Identified From TAM.** ✖*:have the corresponding problem.*

6 Model Limitations

We have shown that TAM is a useful tool for schedulability analysis and delivers promising results. In this section we discuss some of its limitations and how we can extend TAM to further help schedulability analysis.

First, current TAM is best suited for describing SEDA-like systems, where each thread belongs to a specific stage. However, in other concurrency models, threads and stages may not be statically bound. For example, in a run-to-completion model, a single thread may perform multiple tasks until a request is completed, and be scheduled (possibly by yielding) before each task. In this case, a stage would be better defined as the execution between scheduling points, allowing one thread to cross multiple stages. We leave extending TAM to other concurrency models to future work.

Second, various workarounds can be used to mitigate the effects of scheduling problems; most of them involve coordination among stages or predicting workload characteristics. Encoding these mechanisms into TAM, possibly in the form of information flow between stages, would allow it to capture the scheduling effects of indirect workarounds.

Finally, even though different systems might possess the same scheduling problems, the difficulty of fixing their problems could vary vastly based on the system's internal structure and code base. Fixing the unknown resource problem directly in HBase requires only separating the short-circuited read processing from the RPC Read stage; fixing this problem in MongoDB, however, requires a major re-structuring of the Worker stage to account for its complex execution paths. TAM is effective in identifying the problems, but does not give many indications on how difficult solving these problems would be; systematically reasoning about such difficulties is another interesting direction to extend TAM.

7 Related Work

Scheduling as a general problem has been extensively studied in computer science, manufacturing, operational research, and many other fields [34, 52, 53, 60]. Our work differs from the previous ones as we separate the scheduling problem in distributed storage systems into two sub-problems: the meta *schedulability* problem and the specific scheduling problem. For a general-purpose storage system that is designed to work for various workloads and meet various peformance measures, the schedulability problem is answered at the system design/build phase, and concerns whether the system offers proper *scheduling support*: are schedulers placed at the right points in the system and given necessary information and control? Once proper scheduling support is built in (i.e., the system provides schedulability), the user can solve his/her own specific scheduling problem: given her workload, which scheduling policy should she implement on top of the scheduling support provided by the system to realize a particular performance goal?

Such separation distinguishes the TAM approach from other formalization of the scheduling problems, such as queuing networks [13, 32, 41, 59] or stochastic Petri nets [18, 46, 60, 70, 71], which focus on solving specific scheduling problems. For example, traditional queuing network models encode specific scheduling plan information and workload characteristics, and output performance measures [43, 50, 59]. One could view TAM as a queuing network skeleton, stripped of all information but that available at system design time; our schedulability analysis aims to derive properties from the limited information encoded in TAM that would hold after the TAM skeleton is augmented with various workload/queing discipline information to form a complete queuing network. Some techniques developed in the queuing theory context may be borrowed to prove certain properties of the TAM [31, 65]; we leave that as future work.

From a more system-oriented perspective, previous work has focused on proposing scheduling plans that achieve various specific goals [47, 54, 55, 57, 63, 68]. For example, Pisces [55] discusses how to allocate local weights to match client demands and achieve global fairness; Cake [63] proposes a feedback loop to adjust local scheduler behavior to provide latency guarantees; Retro [37] supports different scheduling policies, but by translating these policies into rate limits at local schedulers. All the above works need proper scheduling support to enforce their plans. As current systems usually lack such support (§5), people indeed encouter the five categories of problems we have identified during the realization of their scheduling plans [38, 54, 63, 68]: Mace et al. found that unknown resource usage and blocking prevented them from achieving fairness [38]; Cake [63] had to add scheduling points to HDFS to enforce SLOs. However, in these systems the encountered problems are solved in an ad hoc manner; the solutions are often buried in implementation details or not discussed at all. A general framework that addresses the schedulability problem explicitly and systematically is thus strongly called for.

Monotasks [44] advocates an architecture in which jobs are broken into units of work that each use a single resource, and each resource is managed with a dedicated scheduler. From the TAM perspective, such an architecture eliminates the unknown resource usage and the hidden contention problem, allowing the system to provide better schedulability. The authors indeed observe that this architecture "allows MonoSpark to avoid resource contention and under utilization", as predicted by TAM.

Our work is also similar to SEDA [64] and Flash [45] in the sense that it studies and modifies the thread structure and interactions to improve system performance. Like our work, Capriccio [61] automatically deduces a flow graph and places scheduling points at the graph nodes for thread scheduling.

8 Conclusions

With sharing being one of the key aspects of modern scalable storage systems, correct and flexible scheduling becomes a central goal in system design. To ensure scheduling works as desired, schedulability analysis should be included as an integrated part of the concurrency architecture. The thread architecture model provides a systematic way of performing such analysis, thus turning the art of enabling effective scheduling into a science that is easily accessible and automatable. The software for schedulability analysis (e.g., TADalyzer and the TAM-based simulation framework) is available at http://research.cs.wisc.edu/adsl/Software/TAM.

Acknowledgements

We thank Rebecca Isaacs (our shepherd), the members of ADSL, and the anonymous reviewers for their feedback. We are especially grateful to the reviewer who asked for the model behind the thread architecture diagrams. We thank Michael Gleicher for his guidance in the thread architecture visualization. CloudLab [48] provided infrastructure to run our experiments. This material was supported by funding from NSF grants CNS-1421033, CNS-1218405, and CNS-1838733, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, NetApp, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[2] Byteman Homepage. http://byteman.jboss.org/.

[3] HBase Issue Tracking Page. https://issues.apache.org/jira/browse/HBASE-8884, July 2013.

[4] Cassandra Issues: Move away from SEDA to TPC. https://issues.apache.org/jira/browse/CASSANDRA-10989, January 2016.

[5] HBase Issue Tracking Page. https://issues.apache.org/jira/browse/HBASE-8836, July 2016.

[6] MongoDB Issue Tracking Page. https://jira.mongodb.org/browse/SERVER-24661, July 2016.

[7] MongoDB Issue Tracking Page. https://jira.mongodb.org/browse/SERVER-20328, July 2016.

[8] ANANDKUMAR, A., BISDIKIAN, C., AND AGRAWAL, D. Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints. In *ACM SIGMETRICS Performance Evaluation Review* (2008), vol. 36, ACM, pp. 133–144.

[9] ANGEL, S., BALLANI, H., KARAGIANNIS, T., O'SHEA, G., AND THERESKA, E. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI* (2014), pp. 233–248.

[10] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[11] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18.

[12] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)* (Vancouver, Canada, December 2010).

[13] BOXMA, O. J., KOOLE, G. M., AND LIU, Z. *Queueing-Theoretic Solution Methods for Models of Parallel and Distributed Systems*. Centrum voor Wiskunde en Informatica, Department of Operations Research, Statistics, and System Theory, 1994.

[14] CHEN, Y.-Y. M., ACCARDI, A. J., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. Path-Based Failure and Evolution Management.

[15] CHODOROW, K. *MongoDB: the Definitive Guide*. " O'Reilly Media, Inc.", 2013.

[16] CLARK, D. D., JACOBSON, V., ROMKEY, J., AND SALWEN, H. An analysis of TCP processing overhead. *IEEE Communications magazine 27*, 6 (1989), 23–29.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[18] DAVIDRAJUH, R. Activity-Oriented Petri Net for scheduling of resources. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on* (2012), IEEE, pp. 1201–1206.

[19] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[20] DIWAKER GUPTA, L. C., GARDNER, R., AND VAHDAT, A. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)* (Melbourne, Australia, Nov 2006).

[21] ELSER, J. Personal Correspondence, April 2017.

[22] FINK, B. Distributed Computation on Dynamo-Style Distributed Storage: Riak Pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop* (2012), ACM, pp. 43–50.

[23] FINK, B. Personal Correspondence, April 2017.

[24] GEORGE, L. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. " O'Reilly Media, Inc.", 2011.

[25] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003), pp. 29–43.

[26] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI* (2011), vol. 11, pp. 24–24.

[27] GU, D. Personal Correspondence, April 2017.

[28] GULATI, A., AHMAD, I., WALDSPURGER, C. A., ET AL. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST* (2009), vol. 9, pp. 85–98.

[29] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 13–24.

[30] KAY, J., AND PASQUALE, J. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *ACM SIGCOMM Computer Communication Review* (1993), vol. 23, ACM, pp. 259–268.

[31] KINGMAN, J. The heavy traffic approximation in the theory of queues. In *Proceedings of the Symposium on Congestion Theory* (1965), no. 2, University of North Carolina Press, Chapel Hill, NC.

[32] KLEINROCK, L. *Queueing Systems, Volume 2: Computer Applications*, vol. 66. wiley New York, 1976.

[33] KLOPHAUS, R. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming* (2010), ACM, p. 14.

[34] KOLISCH, R., AND SPRECHER, A. PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research 96*, 1 (1997), 205–216.

[35] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ET AL. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 1–14.

[36] LAKSHMAN, A., AND MALIK, P. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (Big Sky Resort, Montana, Oct 2009).

[37] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *NSDI* (2015), pp. 589–603.

[38] MACE, J., BODIK, P., MUSUVATHI, M., FONSECA, R., AND VARADARAJAN, K. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 144–159.

[39] Matloff, N. Introduction to Discrete-Event Simulation and the SimPy Language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August 2* (2008), 2009.

[40] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter* (1993), vol. 93.

[41] Menga, G., Bruno, G., Conterno, R., and Dato, M. Modeling FMS by closed queuing network analysis methods. *IEEE transactions on components, hybrids, and manufacturing technology 7*, 3 (1984), 241–248.

[42] Muthukkaruppan, K. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)* (Pacific Grove, California, October 2011).

[43] Nguyen, H., Shen, Z., Gu, X., Subbiah, S., and Wilkes, J. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *ICAC* (2013), vol. 13, pp. 69–82.

[44] Ousterhout, K., Canel, C., Ratnasamy, S., and Shenker, S. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 184–200.

[45] Pai, V. S., Druschel, P., and Zwaenepoel, W. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference, General Track* (1999), pp. 199–212.

[46] Peterson, J. L. Petri Nets. *ACM Comput. Surv. 9*, 3 (September 1977), 223–252.

[47] Reda, W., Canini, M., Suresh, L., Kostić, D., and Braithwaite, S. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 95–110.

[48] Ricci, R., Eide, E., and Team, C. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE 39*, 6 (2014), 36–38.

[49] Rizzo, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[50] Schroeder, B., Wierman, A., and Harchol-Balter, M. Open Versus Closed: A Cautionary Tale.

[51] Serenyi, D. Cluster-Level Storage at Google. www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf, 2017.

[52] Shih, H. M., and Sekiguchi, T. A timed Petri net and beam search based online FMS scheduling system with routing flexibility. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on* (1991), IEEE, pp. 2548–2553.

[53] Shirazi, B. A., Kavi, K. M., and Hurson, A. R. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[54] Shue, D., and Freedman, M. J. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 17.

[55] Shue, D., Freedman, M. J., and Shaikh, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI* (2012), vol. 12, pp. 349–362.

[56] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)* (Incline Village, Nevada, May 2010).

[57] Suresh Puthalath, L. On predictable performance for distributed systems.

[58] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 182–196.

[59] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. An Analytical Model for Multi-Tier Internet Services and Its Applications. In *ACM SIGMETRICS Performance Evaluation Review* (2005), vol. 33, ACM, pp. 291–302.

[60] Van der Aalst, W. M. The Application of Petri Nets to Workflow Management. *Journal of circuits, systems, and computers 8*, 01 (1998), 21–66.

[61] von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003), pp. 268–281.

[62] Wachs, M., Abd-El-Malek, M., Thereska, E., and Ganger, G. R. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)* (San Jose, California, February 2007).

[63] Wang, A., Venkataraman, S., Alspaugh, S., Katz, R., and Stoica, I. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), ACM, p. 14.

[64] Welsh, M., Culler, D., and Brewer, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Banff, Canada, October 2001).

[65] Whitt, W. A Light-Traffic Approximation for Single-Class Departure Processes from Multi-Class Queues. *Management science 34*, 11 (1988), 1333–1346.

[66] Xu, Y., and Zhao, M. IBIS: Interposed Big-Data I/O Scheduler. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (2016), ACM, pp. 111–122.

[67] Yang, S. *Schedulability in Local and Distributed Storage Systems*. The University of Wisconsin-Madison, 2017.

[68] Zeng, J., and Plale, B. Workload-Aware Resource Reservation for Multi-tenant NoSQL. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on* (2015), IEEE, pp. 32–41.

[69] Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., and Stumm, M. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *OSDI* (2014), vol. 14, pp. 629–644.

[70] Zhou, M., and DiCesare, F. Parallel and Sequential Mutual Exclusions for Petri Net Modeling of Manufacturing Systems with Shared Resources. *IEEE Transactions on Robotics and Automation 7*, 4 (1991), 515–527.

[71] Zhou, M., and Wu, N. *System Modeling and Control with Resource-Oriented Petri Nets*, vol. 35. Crc Press, 2009.

[72] Zhu, T., Tumanov, A., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.