

Schedulability in Local and Distributed Storage Systems

By

Suli Yang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: September 8th, 2017

The dissertation is approved by the following members of the Final Oral
Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Aditya Akella, Professor, Computer Sciences

Michael M. Swift, Associate Professor, Computer Sciences

Lisa L. Everett, Professor, Physics

All Rights Reserved

© Copyright by Suli Yang 2017

To my advisors

Acknowledgments

During my first years in the graduate school, I was naive, clueless and depressed, without the slightest idea about what a Ph.D is like and what I want to do with my life. Then I met Remzi in an undergraduate operating systems class. He is a great lecturer, but more importantly, he always seems so happy, content and enjoying his life. I decided that if I do the same thing as he does, maybe I would be as happy as he is. That is how I started working with Remzi and later Andrea on my Ph.D in computer sciences.

Of course life is not that simple; I did not get the panacea for all problems in my life. Indeed, getting a Ph.D is more difficult than I expected and I hit several down points during the process. However, looking back, I think working with Andrea and Remzi is still one of the best decisions I have ever made in my life. Both of them are the kind of professors who put their students first and offer their help unconditionally. I was not always an easy student to work with. When I was going nowhere in my research, when I failed to even show up and deliver anything, when I kept rejecting their suggestions, they did not abandon me. Instead they always try to help me in any ways they can, even when I was constantly failing them. This kind of unconditional support is something I never experienced before (or even knew existed), and it changes my life in a profound way.

I also learn so much by just observing them: their light-hearted attitude to life, their empathy to others, how they prioritize, and their strong relationship between each other. Their wisdom just sparkles in every corners of life. I can still quote Remzi, "being smart is a small gift", and Andrea, "things do not get easier".

On top of that, they are also the best academic advisors a student can ask for. Remzi has the amazing ability to simplify complex material into clear concepts, while Andrea does wonders when organizing seemingly scattered ideas into compelling, coherent stories. However, these seem almost secondary when compared to how they inspired me in my life. I thus extend my greatest thanks to my advisors, Andrea and Remzi Arpacı-Dusseau, and dedicate this dissertation to them.

I am fortunate to spend my Ph.D years in UW-Madison with so many brilliant faculty members; going to their classes often feel like a blessing because of the exciting intellectual engagement one can expect. In particular, I thoroughly enjoyed Lisa Everett's elegant development of quantum mechanics in PHY731, Aditya Akella's passionate discussion on SDN researches in CS838 and Dieter van Melkebeek's rigorous analysis of algorithms in CS787. I would also like to thank Sau Lan Wu, my former advisor in Physics department, for supporting me during my first year in graduate school and introducing me to several professors when I decided to switch to computer sciences.

Interning at NetApp has benefited me in many ways; I feel very lucky to have the opportunity to work in the Advanced Technology Group with many great mentors and colleagues. I would like to thank Kiran Srinivasan, Swetha Krishnan, Kishore Udayashankar, Jingxin Feng and Sethuraman Subbiah for being great mentors and Siva Jayasenan for being a terrific manager. They are always around to answer my questions despite how busy they are, and they are incredibly understandable and supportive when my depression gets in the way of my work.

I learned a great deal from my peers at UW-Madison and it is truly a privilege to work with them. It would not have been possible for me to finish the Split-I/O project without the help of Tyler Harter. He acted as a mentor to me and taught me many things about research, from meeting advisors in an effective way to visualizing data with the least ink possible. He is also a wonderful friend who I feel I can confide in. Jing Liu is a pleasure to work with; her attention to details complements me and improves our project in ways that I cannot. Zev Weiss helped proof-reading our paper under a tight timeline and offered invaluable advice. I also thank Ram Alagappan, Yuvraj Patel, Jun He, Sudarsun Kannan, Lanyue Lu, Samer Al-Kiswany, Thanumalayan Pillai, Aishwarya Ganesan, Leo Arulraj, Yupu Zhang, Yiying Zhang and many other students in the Arpaci-Dusseau group for their insightful discussion and feedbacks.

Many good friends have been a great support to me during my PhD. I especially want to thank Peisi Huang for being there during the good and bad times of my life; she is truly a friend in need. I thank Jinlu Miao, Siyuan Zhang and Yijin Wang for being wonderful roommates and friends; I still smile when looking at gifts from them. I thank Linhai Song, Wenfei Wu, Linqiao Qin and Sihui Yang for their friendship and the many meals they shared with me. Finally, I thank Yushen Li and Emir Habul for the good time we shared and the life lessons I learned from them.

Contents

Acknowledgments	ii
Contents	v
Abstract	ix
1 Introduction	1
1.1 Scheduling in Local Storage Systems	5
1.2 Scheduling in Distributed Storage Systems	8
1.3 Overview	10
2 Split-Level I/O Scheduling on Local File Systems	11
2.1 Background	12
2.1.1 The Storage Stack	12
2.1.2 Framework Architectures	14
2.2 Motivation	15
2.2.1 Framework Support for Schedulers	16
2.2.2 File-System Challenges	18
2.3 Split Framework Design	25
2.3.1 Cause Mapping	26
2.3.2 Cost Estimation	27
2.3.3 Reordering	28

2.4	Split Scheduling in Linux	29
2.4.1	Cross-Layer Tagging	30
2.4.2	Scheduling Hooks	30
2.4.3	Implement a Split-Level Scheduler	34
2.4.4	Implementation Effort and Overhead	34
2.5	Scheduler Case Studies	36
2.5.1	AFQ: Actually Fair Queuing	37
2.5.2	Deadline	39
2.5.3	Token Bucket	41
2.5.4	Implementation Effort	47
2.6	File System Integration	47
2.7	Real Applications	49
2.7.1	Databases	50
2.7.2	Virtual Machines (QEMU)	53
2.7.3	Distributed File Systems (HDFS)	54
2.8	Conclusion	56
3	Thread Architecture Diagrams	58
3.1	Thread Architecture Diagrams	59
3.2	TADalyzer	62
3.2.1	Automatic Discovery	62
3.2.2	Limitations	63
3.3	HBase/HDFS	64
3.3.1	Request Flow	66
3.3.2	Summary	68
3.4	MongoDB	68
3.4.1	Request Flow	68
3.4.2	Summary	70
3.5	Cassandra	70
3.5.1	Request Flow	72
3.5.2	Summary	73

3.6	Riak KV	73
3.6.1	Request Flow	73
3.6.2	Summary	75
3.7	Conclusions	75
4	Maat: Toward Schedulability on Distributed Storage Systems	76
4.1	TAD Simulation	78
4.1.1	Design	78
4.1.2	Simplifications	81
4.2	The Maat Principle	82
4.2.1	No Scheduling Points	82
4.2.2	Unknown Resource Usage	85
4.2.3	Hidden Contention	87
4.2.4	Blocking	91
4.2.5	Ordering Constraint	92
4.2.6	Summary and Discussion	94
4.3	Applying Maat to HBase	96
4.3.1	No Scheduling	97
4.3.2	Unknown Resource Usage	98
4.3.3	Hidden Contention	100
4.3.4	Blocking	101
4.3.5	Ordering Constraints	103
4.4	HBase-Maat Implementation	106
4.4.1	Implementation Experience	106
4.4.2	TAD Validation	107
4.5	Analyzing Other TADs	112
4.5.1	MongoDB	112
4.5.2	Cassandra	117
4.5.3	Riak	121
4.5.4	Discussions	125
4.6	Conclusions	126

5	Related Work	127
5.1	Scheduling in Local Storage System	127
5.2	Scheduling in Distributed Storage Systems	130
6	Conclusions and Future Work	133
6.1	Summary	133
6.2	Lessons Learned	134
6.3	Future Work	136
6.3.1	Scalable Scheduling Architecture	136
6.3.2	Completeness of the Maat Principle: A Formal Analysis	138
6.3.3	Build a Natively Schedulable Storage System	139
6.4	Closing Words	140
	Bibliography	141

Abstract

Scheduling is the key-enabler in any resource sharing systems, be it the operating system or the shared data center. By controlling which client or application is serviced, critical features including fair sharing, throughput guarantees, low tail latency and performance isolation can be successfully realized. The storage stack, which provides persistence of data, is one of the most important components in almost all systems. However, scheduling in storage stacks has largely remained an unsolved problem; existing storage systems offer very weak, if any, performance guarantees. In this dissertation, we look at the scheduling problem in modern, multi-layer storage systems, including the local and distributed storage stacks.

For the local storage stack, we first demonstrate that traditional block-level I/O schedulers are unable to meet throughput, latency and isolation goals. To overcome the limitations of traditional scheduling frameworks, we introduce *split-level I/O scheduling*, a new framework that splits I/O scheduling logic across handlers at three different layers: block, system call, and page cache. By utilizing the split-level framework, we build a variety of novel schedulers to readily achieve these goals: our Actually Fair Queuing scheduler reduces priority-misallocation by $28\times$; our Split-Deadline scheduler reduces tail latencies by $4\times$; our Split-Token scheduler reduces sensitivity to interference by $6\times$. We show that the framework is general and operates correctly with disparate file systems (ext4 and XFS).

Finally, we demonstrate that split-level scheduling serves as a useful foundation for databases (SQLite and PostgreSQL), hypervisors (QEMU), and distributed file systems (HDFS), delivering improved isolation and performance in these important application scenarios.

Effective scheduling in existing distributed storage systems has remained difficult despite repeated attempts from both industry and academia; these systems usually provide weak or no performance guarantees. We introduce Maat, an approach to systematically examining the schedulability of a system, identifying its scheduling problems and enabling effective scheduling in the system. Following the Maat approach, We use *Thread Architecture Diagrams (TADs)* to describe the behavior and interactions of different threads in a system, and show both how to construct TADs for existing systems as well as utilize TADs to identify critical scheduling problems. We identify five fundamental problems that prevent a system from being Maat-adherent and show that these problems arise in existing systems, making it difficult or impossible to realize various scheduling goals. By applying the Maat guideline, we derive HBase-Maat, a flexible scheduling architecture for HBase that can realize the desired scheduling disciplines even when presented with challenging workloads.

1

Introduction

Resource sharing has always been one of the central problems in computer science, and it is becoming a theme of many technical trends going on today [49, 53, 96]. Operating systems are built around the idea of sharing computer hardware and software resources among different programs [21, 38, 75]. Virtualization extends this idea further, enabling hardware resources to be shared among different virtual machines, each running its own operating system [30, 70, 115]. Such sharing across operating systems has had abundant success in recent years [119]. Software containers, such as Docker [87] and Linux Kernel Containment (LXC) [97], present a different way to achieve similar level of sharing as virtual machines. Instead of running multiple operating systems on the same hardware, the kernel of the operating system is equipped to allow multiple isolated user-space instances, and each instance looks like a full OS environment from the user's point of view.

Cloud computing has brought us a new economical model of resource sharing; the cloud vendor provides a pool of computing resources in the data centers, including hardware, networking, storage and softwares, and other parties utilize these resources on demand with minimal management effort [23, 86]. Cloud computing relies on sharing of resource to achieve coherence and economy of scale. With significant capital and operational benefits, the cloud computing model is becoming prevalent in

recent years. It reached \$209 billion revenue in 2016, and is projected to affect more than \$1 trillion in IT spendings by 2020 [42].

Scheduling is the key-enabler in any resource-sharing systems, be it the operating system, virtual machine monitor or the shared data center. Through careful scheduling, a single resource can be multiplexed between multiple clients, creating the illusion that each client has its own resource. By controlling which client or application is serviced, critical features including fair sharing [55, 80, 81, 109, 123], throughput guarantees [102, 128], low tail latency [39, 56, 117, 130] and performance isolation [19, 103, 114] can be successfully realized.

In this dissertation we focus on the sharing of *storage resources* and the scheduling problems arising from it. We broadly classify storage as the hardware and software used to provide *persistence* of data, despite computer crashes or power outages. The explosion of data volume, the magnification of data criticality, and the increasing dependency of businesses on big data in recent years have made storage one of the most important computing resources. In the meantime, storage resource scheduling and isolation have largely been an unsolved problem due to some unique challenges that are not present when scheduling other resources, such as CPU or network. First, storage scheduling decisions have both short term and long term ramifications. For example, when a scheduler makes a decision of where to write data, it will affect performance later when data is being read back. Second, the overall capacity of the storage system is affected by the workload running on top of it and the scheduling decisions being made. Finally, the consistency constraints associated with data usually limits the flexibility of scheduling.

The aforementioned challenges make it a difficult task to schedule storage resources. This task is further complicated by the fact that modern storage systems are often quite complex and have different components interacting with each other. One question naturally arises: *where should we*

place scheduling logic in such complex systems to effectively meet our scheduling goals? The answer to this question is not as obvious as it may seem like, because the interactions between different components of the system can have unexpected effects on scheduling, making it hard to enforce scheduling disciplines at any given place in the system. For example, information critical in making scheduling decisions may get lost in interactions between components; or one component could impose artificial limitations on scheduling done at other components, leading to poor performance of the system. We have observed both phenomena in commonly-used systems, and they can affect performance by orders of magnitude.

Previous works [31, 62, 89, 100] on scheduling in storage systems have mostly focused on optimizations within a single layer or component in the system. Not much thought was given to *why* scheduling is placed at this particular layer/component, or how the rest of the system could affect scheduling. These kind of approaches have led to poor performance and failure to meet scheduling goals, both in local and distributed storage systems. For example, traditionally I/O scheduling for a local host has always been done at the block level, but as we will show later (§2.2.2), the file system running on top of the block layer has a major impact on I/O scheduling, and can render scheduling decisions made at the block layer useless. Such *block-level scheduling* causes long I/O latency, unfairness in I/O resource allocation, and failure to isolation.

More recent work, such as Pisces [103], PriorityMeister [130] and Redline [126], have taken a more holistic view of the storage system and recognized the necessity to schedule at multiple points in the system. For example, Redline [126] tries to avoid unresponsiveness in the local storage system by scheduling at both the buffer cache level and the block level; Pisces [103] aims at providing isolation and fairness for multi-tenant cloud storage by combining proxy-level replica selection, node-level weight allocation and global-level data migration. However, these projects are

mostly concerned about how to realize a particular scheduling goal or policy (e.g., fairness), and achieve it by carefully tuning the scheduling algorithms. We, on the other hand, believe that a few basic mechanisms could be built into the complex storage stack so that one could easily realize *any* scheduling policy she has in mind, be it fairness, isolation, meeting of deadline, or others.

In this dissertation, we examine how we could build such scheduling mechanisms in modern complex storage systems. We first look at the local storage stack in a single computer node. We show that current approach, namely the block-level I/O scheduling, provides insufficient mechanisms to realize important I/O scheduling policies. The building of local storage scheduling mechanisms leads to *split-level I/O scheduling*, a novel scheduling framework in which a scheduler is constructed across several layers. By implementing a judiciously selected set of handlers at key junctures within the storage stack (namely, at the system-call, page-cache, and block layer), a developer can implement a scheduling discipline with full control over behavior of different layers and with no loss in high- or low-level information. We demonstrate the generality of split scheduling by implementing three new schedulers: AFQ (Actually-Fair Queuing) provides fairness between processes, Split-Deadline observes latency goals, and Split-Token isolates performance, and show vast improvements (6x - 28x) over similar schedulers in other frameworks.

We then study the scheduling problem of distributed storage systems. We investigate popular distributed storage systems including HBase/HDFS, MongoDB, and Riak. We identify five fundamental scheduling problems in these systems: a lack of local scheduling control points, unknown resource usage, hidden competition between threads, uncontrolled thread blocking, and ordering constraints upon requests. These problems are commonly present in widely-used storage systems, and make it difficult or impossible to realize scheduling disciplines. We introduce *Thread Ar-*

chitecture Diagrams (TADs) to describe the behaviors and interactions of different threads in a system, and show both how to construct TADs for existing systems as well as utilize TADs to identify critical scheduling problems. We introduce Maat, an approach to solve the scheduling problems and provide effective scheduling. By applying the Maat guidelines, we derive HBase-Maat, a flexible scheduling architecture for HBase that can realize the desired scheduling disciplines even when presented with challenging workloads.

The central finding of this dissertation is that *schedulers need proper support*. A scheduler can only be effective when it is placed at the right place of the system, given the right information and empowered of enough control to reorder the requests. In all system we studied, we found ineffective scheduling not because of inadequate scheduling algorithm, but insufficient support to the scheduler from the system. This dissertation is thus about how to add scheduling support to various existing systems to enable different scheduling policies; it also suggests design principles to make future systems more scheduling friendly.

1.1 Scheduling in Local Storage Systems

Deciding which I/O request to schedule, and when, has long been a core aspect of the operating system storage stack [18, 21, 62–64, 79, 91, 99–101, 122]. Each of these approaches has improved different aspects of I/O scheduling; for example, research in single-disk schedulers incorporated rotational awareness [63, 64, 100]; other research tackled the problem of scheduling within a multi-disk array [121, 127]; more recent work has targeted flash-based devices [68, 89], tailoring the behavior of the scheduler to this new and important class of storage device. All of these optimizations and techniques are important; in sum total, these systems can improve overall system performance dramatically [54, 100, 127] as

well as provide other desirable properties (including fairness across processes [26] and the meeting of deadlines [126]).

Most I/O schedulers (hereafter just “schedulers”) are built at the block level within an operating system, beneath the file system and just above the device itself. Such *block-level* schedulers are given a stream of requests and are thus faced with the question: which requests should be dispatched, and when, in order to achieve the goals of the system?

Unfortunately, making decisions at the block level is problematic, for two reasons. First, and most importantly, the block-level scheduler fundamentally cannot reorder certain write requests; file systems carefully control write ordering to preserve consistency in the event of system crash or power loss [45, 59]. Second, the block-level scheduler cannot perform accurate accounting; the scheduler lacks the requisite information to determine which application was responsible for a particular I/O request. Due to these limitations, block-level schedulers cannot implement a full range of policies.

An alternate approach, which does not possess these same limitations, is to implement scheduling much higher in the stack, namely with system calls [37]. *System-call scheduling* intrinsically has access to necessary contextual information (*i.e.*, which process has issued an I/O). Unfortunately, system-call scheduling is no panacea, as the low-level knowledge required to build effective schedulers is not present. For example, at the time of a read or write, the scheduler cannot predict whether the request will generate I/O or be satisfied by the page cache, information which can be useful in reordering requests [20, 112]. Similarly, the file system will likely transform a single write request into a series of reads and writes, depending on the crash-consistency mechanism employed (*e.g.*, journaling [59] or copy-on-write [98]); scheduling without exact knowledge of how much I/O load will be generated is difficult and error prone.

We introduce *split-level I/O scheduling*, a novel scheduling framework

in which a scheduler is constructed across several layers. By implementing a judiciously selected set of handlers at key junctures within the storage stack (namely, at the system-call, page-cache, and block layers), a developer can implement a scheduling discipline with full control over behavior and with no loss in high- or low-level information. Split schedulers can determine which processes issued I/O (via graph tags that track causality across levels) and accurately estimate I/O costs. Furthermore, memory notifications make schedulers aware of write work as soon as possible (not tens of seconds later when writeback occurs). Finally, split schedulers can prevent file systems from imposing orderings that are contrary to scheduling goals.

We demonstrate the generality of split scheduling by implementing three new schedulers: AFQ (Actually-Fair Queuing) provides fairness between processes, Split-Deadline observes latency goals, and Split-Token isolates performance. Compared to similar schedulers in other frameworks, AFQ reduces priority-misallocation errors by $28\times$, Split-Deadline reduces tail latencies by $4\times$, and Split-Token improves isolation by $6\times$ for some workloads. Furthermore, the split framework is not specific to a single file system; integration with two file systems (ext4 [83] and XFS [107]) is relatively simple.

Finally, we demonstrate that the split schedulers provide a useful base for more complex storage stacks. Split-Token provides isolation for both virtual machines (QEMU) and data-intensive applications (HDFS), and Split-Deadline provides a solution to the database community's "fsync freeze" problem [3, 11, 12]. In summary, we find split scheduling to be simple and elegant, yet compatible with a variety of scheduling goals, file systems, and real applications.

1.2 Scheduling in Distributed Storage Systems

Modern distributed storage systems are complex, concurrent programs. Many systems are realized via an intricate series of stages, queues, and thread pools, based loosely on the SEDA design principles [120]. Understanding how to introduce scheduling control into these systems is challenging; a single request may flow through numerous stages across multiple machines before being completed.

To address the scheduling problem in the highly concurrent distributed storage systems, we introduce Maat, an approach to systematically examining the *schedulability* of a system, identifying its scheduling problems, and enabling scheduling control in the system. The Maat principle, named after the Egyptian concept of order, specifies three conditions that make realizing a scheduling policy easy:

Completeness – the system provides necessary scheduling points so that a global policy *can* be translated into local scheduling plans at these points.

Local enforceability – the local scheduling plans *can* be implemented. At each scheduling point, the system provides both enough *information* and *control* to the local scheduler to make implementing the plan possible.

Independent scheduling – the decisions made by one local scheduler do not have unexpected effects at other scheduling points.

Following the Maat principle, we first demonstrate a method to discover the schedulability of these systems. Our method traces a system of interest under various workloads and produces a *Thread Architecture Diagram (TAD)*; by analyzing a TAD, scheduling problems can be discerned, pointing towards solutions that introduce necessary scheduling controls. We produce TADs for four important and widely-used scalable storage systems: HBase/HDFS [46, 104], Cassandra [73], MongoDB [34], and Riak [71], and highlight weaknesses in each systems' scheduling architecture. Our analysis centers around five common problems that cause violation of the Maat conditions and in turn lead to inadequate schedul-

ing: a lack of local scheduling control points, unknown resource usage, hidden competition between threads, uncontrolled thread blocking, and ordering constraints upon requests.

The solutions to overcome these difficulties, and thus enable scheduling control in existing systems, are based on two core classes of techniques. The first are *direct* methods, which explicitly alter the existing thread architecture to avoid a specific scheduling problem; direct techniques can sometimes be more challenging to implement, depending on the exact concurrency architecture. The second are *indirect* methods, which use information to overcome scheduling limitations; indirect approaches [80, 117, 130] are easier to incorporate but more approximate in the scheduling control they provide.

To show the benefits of the Maat approach, we apply it to the most complex system that we studied, HBase/HDFS. We first identify its scheduling problems; then through a combination of direct and indirect methods, we show how HBase-Maat can be transformed to provide schedulability. Specifically, HBase-Maat improves performance (by a factor of 3) under intense resource competition and thus enables fair-sharing of system throughput; HBase-Maat also significantly improves performance under mixed workloads (sometimes by a factor of 50) by enabling cached (fast) requests to be fairly serviced; finally, HBase-Maat achieves proper isolation despite variances in request amount, size, and other workload factors. Although we utilize implementation to demonstrate the ultimate efficacy of Maat, we also show how targeted simulations are useful in schedulability analysis, especially when considering alternatives. To this end, we also build a simulation framework to facilitate the scheduling study in other SEDA-based systems.

1.3 Overview

We now briefly describe the contents of the different chapters in this dissertation.

In Chapter 2 we discuss in detail the scheduling problems in the local storage stack, especially in the presence of modern file systems. We then introduce the split-level I/O framework to solve these problems and enable the developers to implement a full range of scheduling policy.

In Chapter 3 we demonstrate how to produce a *Thread Architecture Diagram* (TAD) to discover the schedulability of a distributed storage system. We also study the scheduling behavior of four popular systems, namely HBase/HDFS, Cassandra, MongoDB and Riak under the lens of TAD.

In Chapter 4 we introduce the Maat scheduling principle. Using the Maat principle, we identify five scheduling problems that commonly arise in distributed storage systems, and general solutions to solve these problems and add scheduling control into existing systems.

In Chapter 5 we summarize previous effort people made toward scheduling in storage systems and discuss how they are related to this dissertation.

Chapter 6 summarizes our studies and highlights some general lessons we learn. We also discuss possible future works here.

2

Split-Level I/O Scheduling on Local File Systems

Deciding which I/O request to schedule, and when, has long been a core aspect of the operating system storage stack [18, 21, 62–64, 79, 91, 99–101, 122]. In this chapter, we first show the traditional block-level and syscall-level scheduling are problematic and cannot implement a full range of policies. We then introduce *split-level I/O scheduling*, a novel scheduling framework in which a scheduler is constructed across several layers. The split-level scheduling framework enables a developer to implement a scheduling discipline with full control over behavior and with no loss in high- or low-level information.

The rest of this chapter is organized as follows. We first give some background on how I/O scheduling works and the role a scheduling framework plays (§2.1). We then evaluate existing frameworks and describe the challenges they face (§2.2). Next we discuss the principles of split scheduling (§2.3) and our implementation in Linux (§2.4). We then implement three split schedulers as case studies (§2.5), discuss integration with other file systems (§2.6), and evaluate our schedulers with real applications (§2.7). Finally, we conclude (§2.8).

2.1 Background

In this section we give some background information on the structure of the local storage stack and how different types of I/O scheduling frameworks operate.

2.1.1 The Storage Stack

Schedulers allocate disk I/O to processes, but processes do not typically use hard disks or SSDs directly. Instead, there is a complex storage stack sitting in between the applications and the storage device. We use Linux as an example here to introduce the basic structure of local storage stacks; the storage stacks in FreeBSD [95] and other operating systems [8] are similar.

As shown in Figure 2.1, the first component with which Linux programs (processes) interact when processing data is the virtual file system (VFS); through VFS the process could invoke the same generic system calls (`open()`, `read()`, `write()`, etc.) to access file data for different file systems on different media.

VFS encapsulates a variety of individual file systems implementations, including ext4 [7], XFS [107], Btrfs [82], and others [24, 29, 111]. Individual file systems implement the generic VFS methods, translating file accesses into block I/Os; they typically use the page cache to speed up the data accesses. Complex mechanisms are implemented within individual file systems to improve performance or to ensure consistency; for example, some file systems may delay allocating disk space for a new file to opportunistically optimize the disk layout, or use journal transactions to log file updates. As we will see later (§2.2.2), these mechanisms may cause many difficulties for I/O scheduling.

The block level requests issued by the file systems are sent to the block layer, which then forwards the requests to the storage devices. For Linux,

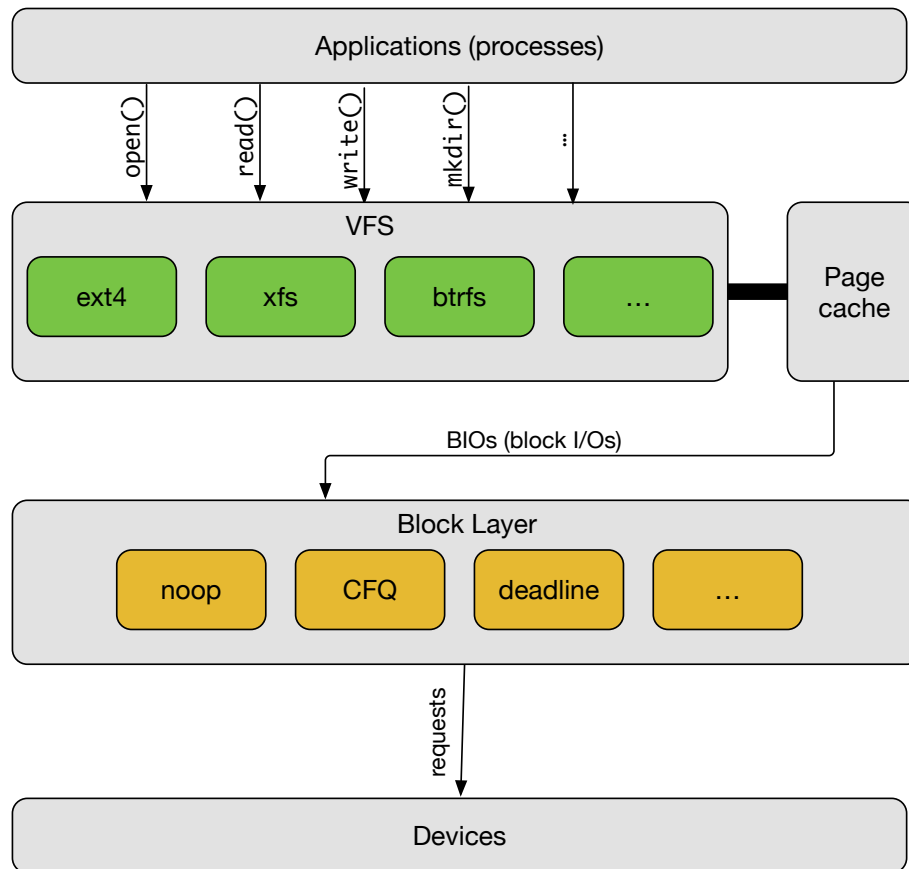


Figure 2.1: **Simplified Linux Storage Stack Diagram.**

which schedules I/O at the block level, different block-level schedulers (e.g., CFQ, deadline) can also be attached at this layer; these schedulers control which requests to dispatch to the underlying devices, and when, to achieve various goals such as fairness or latency guarantees. It is also possible to schedule I/Os at other layers in the storage stack, which we discuss next.

2.1.2 Framework Architectures

Following the old wisdom of separating mechanisms and policies, I/O scheduling in the local storage stack is usually separated into the scheduling framework and individually schedulers. The framework provides a running environment to different schedulers, while an individual scheduler is responsible for realizing a particular scheduling policy (e.g., fairness or latency guarantee).

Scheduling frameworks offer hooks to which individual schedulers can attach; via these hooks, a framework passes information and exposes control to schedulers. Schedulers implement these hooks to achieve various scheduling goals. Different scheduling frameworks provide hooks at different points in the storage stack; we categorize frameworks by the level at which the hooks are available. Figure 2.2 shows the architecture of different types of scheduling frameworks.

Figure 2.2(a) illustrates block-level scheduling, the traditional approach implemented in Linux [10], FreeBSD [95], and other systems [8]. Clients initiate I/O requests via system calls, which are translated to block-level requests by the file system. Within the block-scheduling framework, these requests are then passed to the scheduler along with information describing them: their location on storage media, size, and the submitting process. Based on such information, a scheduler may reorder the requests according to some policy. For example, a scheduler may accumulate many requests in its internal queues and later dispatch them in an order that improves sequentiality.

Figure 2.2(b) shows the system-call scheduling architecture (SCS) proposed by Craciunas *et al.* [37]. Instead of operating beneath the file system and deciding when block requests are sent to the storage device, a system-call scheduler operates on top of the file system and decides when to issue I/O related system calls (read, write, etc.). When a process invokes a system call, the scheduler is notified. The scheduler may put the process to

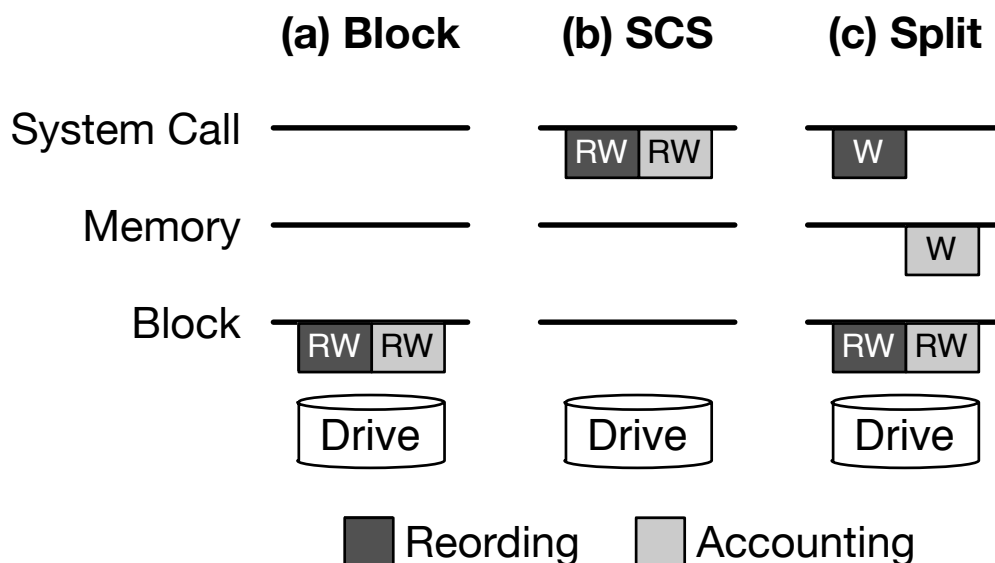


Figure 2.2: **Scheduling Architectures.** *The boxes show where scheduler hooks exist for reordering I/O requests or doing accounting. Sometimes reads and writes are handled differently at different levels, as indicated by “R” and “W”.*

sleep for a time before the body of the system call runs. Thus, the scheduler can reorder the calls, controlling when they become active within the file system.

Figure 2.2(c) shows the hooks of the split framework, which we describe in a later section (§2.4.2). In addition to introducing novel page-cache hooks, the split framework supports select system-call and block-level hooks.

2.2 Motivation

Block-level schedulers are severely limited by their inability to gather information from and exert control over other levels of the storage stack. As an example, we consider the Linux CFQ scheduler, which supports

an `ionice` utility that can put a process in idle mode. According to the man page: “*a program running with idle I/O priority will only get disk time when no other program has asked for disk I/O*” [9]. Unfortunately, CFQ has little control over write bursts from idle-priority processes, as writes are buffered above the block level.

We demonstrate this problem by running a normal process A alongside an idle-priority process B. A reads sequentially from a large file. B issues random writes over a one-second burst. Figure 2.3 shows the result: B quickly finishes while A (whose performance is shown via the CFQ line) takes over five minutes to recover. Block-level schedulers such as CFQ are helpless to prevent processes from polluting write buffers with expensive I/O. As we will see, other file-system features such as journaling and delayed allocation are similarly problematic.

The idle policy is one of many possible scheduling goals, but the difficulties it faces at the block level are not unique. In this section, we consider three different scheduling goals, identifying several shared needs (§2.2.1). Next, we show existing frameworks are fundamentally unable to meet scheduler needs when running in conjunction with a modern file system (§2.2.2).

2.2.1 Framework Support for Schedulers

We now consider three commonly used I/O schedulers: CFQ [2], Deadline [4], and token-bucket [108], identifying what framework support is needed to implement these schedulers correctly.

CFQ: CFQ aims to allocate I/O resources fairly between processes based on their priorities [2]. To do so, CFQ must be able to track which process is responsible for which I/O requests, estimate how much each request costs, and reorder higher-priority requests before lower-priority requests. Other schedulers such as SFQ [50] and YFQ [28] share similar goals with CFQ.

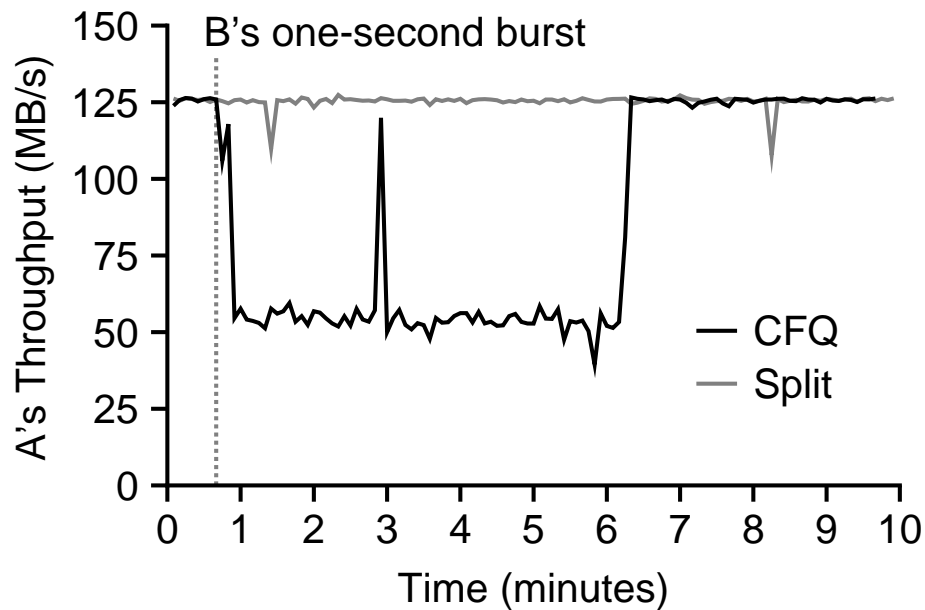


Figure 2.3: **Write Burst.** *B's one-second random-write burst severely degrades A's performance for over five minutes. Putting B in CFQ's idle class provides no help.*

Deadline: The Deadline scheduler observes deadlines for I/O operations, offering predictable latency to applications that need it [4]. The deadline scheduler needs to map an application's deadline setting to each request and issue lower-latency requests before other requests.

Token-Bucket: The token-bucket scheduler caps the resources an application may use, regardless of overall system load [108]. Limits are useful when resources are purchased and the seller does not wish to give away free I/O. The token-bucket scheduler needs to know the cost and causes of I/O operations in order to throttle correctly.

Although the above schedulers have distinct goals, they have three common needs. First, schedulers need to be able to *map causes* to identify which process is responsible for an I/O request. Second, schedulers need to *estimate costs* in order to optimize performance and perform accounting properly. Third, schedulers need to be able to *reorder* I/O requests so that

Scheduler	Goal	Support Needed			
		CM	CE	R	O
CFQ [2]	Fairness	✗	✗	✗	
Deadline [4]	Deadline	✗	✗	✗	
Token-Bucket [108]	Isolation	✗	✗	✗	
QUASIO [66]	Low Response Time			✗	Task-I/O Dependency
BAA [118]	Fairness&Efficiency	✗	✗	✗	
FIOS [89]	Fairness&Efficiency	✗	✗	✗	
Stream Scheduling [124]	High Throughput		✗	✗	
PARDA [55]	Fairness	✗		✗	
EW-Sched [129]	Low Latency		✗	✗	
Freeblocks [78]	High Utilization		✗	✗	
Semi-preemptible IO [40]	Priority Enforcement	✗	✗	✗	
Faccade [77]	Deadline	✗	✗	✗	

Table 2.1: **Framework Support Needed by Different Schedulers.** *CM: Cause Mapping, CE: Cost Estimation, R: Reordering, O: Other support needed.*

the operations most important to achieving scheduling goals are served first.

In fact, these three requirements (cause mapping, cost estimation and reordering) are shared among most schedulers. To illustrate, we list in Table 2.1 the above three schedulers as well as the I/O schedulers proposed in the last 15 years in the USENIX Conference on File and Storage Technologies (FAST), one of the leading conferences on I/O and storage technologies. We could see that providing cause mapping, cost estimation and reordering capacity could sufficiently meet the need of all proposed schedulers except for one. Unfortunately, as we will see, current block-level and system-call schedulers cannot provide all the support schedulers need.

2.2.2 File-System Challenges

As we discuss earlier (§2.1.1), schedulers allocate disk I/O to processes, but processes do not typically use hard disks or SSDs directly. Instead, processes request service from a file system, which in turn translates re-

quests to disk I/O. Unfortunately, file systems make it challenging to satisfy the needs of the scheduler. We now examine the implications of writeback, delayed allocation, journaling, and caching for schedulers, showing how these behaviors fundamentally require a restructuring of the I/O scheduling framework.

2.2.2.1 Delayed Writeback and Allocation

Delayed writeback is a common technique for postponing I/O by buffering dirty data to write at a later time. Procrastination is useful because the work may go away by itself (*e.g.*, the data could be deleted) and, as more work accumulates, more efficiencies can be gained (*e.g.*, sequential write patterns may become realizable).

Some file systems also delay allocation to optimize data layout [83, 107]. When allocating a new block, the file system does not immediately decide its on-disk location; another task will decide later. More information (*e.g.*, file sizes) becomes known over time, so delaying allocation enables more informed decisions.

Both delayed writeback and allocation involve file-system level delegation, with one process doing I/O work on behalf of other processes. A writeback process submits buffers that other processes dirtied and may also dirty metadata structures on behalf of other processes. Such delegation obfuscates the mapping from requests to processes. To block-level schedulers, the writeback task sometimes appears responsible for *all* write traffic.

We evaluate Linux's priority-based block scheduler, CFQ (Completely Fair Queuing) [2], using an asynchronous write workload. CFQ aims to allocate disk time fairly among processes (in proportion to priority). We launch eight threads with different priorities, ranging from 0 (highest) to 7 (lowest): each writes to its own file sequentially. A thread's write throughput should be proportional to its priority, as shown by the ex-

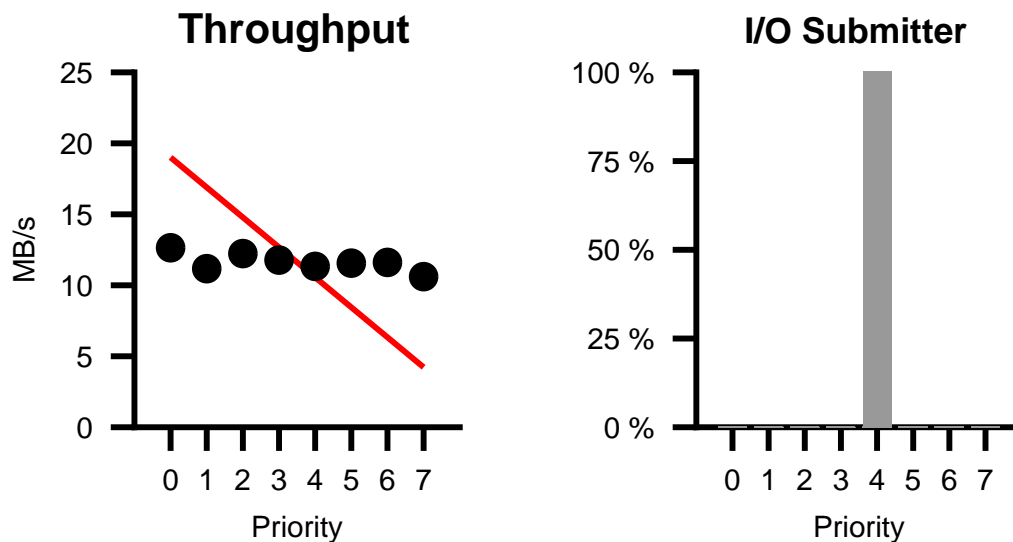


Figure 2.4: **CFQ Throughput.** *The left plot shows sequential write throughput for different priorities. The right plot the portion of requests for each priority seen by CFQ. Unfortunately, the “Completely Fair Scheduler” is not even slightly fair for sequential writes.*

pectation line of Figure 2.4 (left). Unfortunately, CFQ ignores priorities, treating all threads equally. Figure 2.4 (right) shows why: to CFQ all the requests appear to have a priority of 4, because the writeback thread (a priority-4 process) submits all the writes on behalf of the eight benchmark threads.

2.2.2.2 Journaling

Many modern file systems use journals for consistent updates [24, 83, 107]. While details vary across file systems, most follow similar journaling protocols to commit data to disk; here, we discuss ext4’s ordered-mode to illustrate how journaling severely complicates scheduling.

When changes are made to a file, ext4 first writes the affected data blocks to disk, then creates a journal transaction which contains all related metadata updates and commits that transaction to disk, as shown

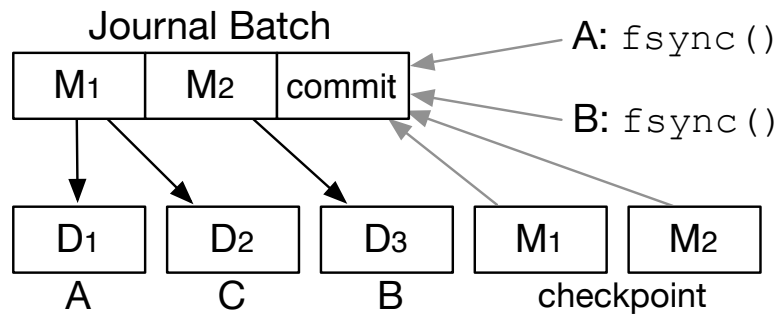


Figure 2.5: **Journal Batching.** Arrows point to events that must occur before the event from which they point. The event for the blocks is a disk write. The event for an `fsync` is a return.

in Figure 2.5. The data blocks (D_1 , D_2 , D_3) must be written before the journal transaction, as updates become durable as soon as the transaction commits, and ext4 needs to prevent the metadata in the journal from referring to data blocks containing garbage. After metadata is journaled, ext4 eventually checkpoints the metadata in place.

Transaction writing and metadata checkpointing are both performed by kernel processes instead of the processes that initially caused the updates. This form of write delegation also complicates cause mapping.

More importantly, journaling prevents block-level schedulers from re-ordering. Transaction batching is a well-known performance optimization [59], but block schedulers have no control over which writes are batched, so the journal may batch together writes that are important to scheduling goals with less-important writes. For example, in Figure 2.5, suppose A is higher priority than B. A's `fsync` depends on transaction commit, which depends on writing B's data. Priority is thus inverted.

When metadata (e.g., directories or bitmaps) is shared among files, journal batching may be necessary for correctness (not just performance). In Figure 2.5, the journal could have conceivably batched M_1 and M_2 separately; however, M_1 depends on D_2 , data written by a process C to a

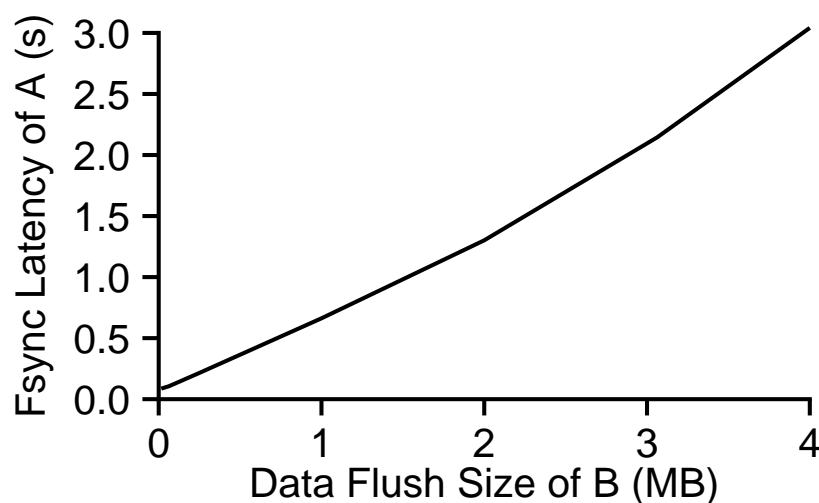


Figure 2.6: **I/O Latency Dependencies.** *Thread A keeps issuing `fsync` to flush one block of data to disk, while thread B flushes multiple blocks using `fsync`. This plot shows how A's latency depends on B's I/O size.*

different file, and thus A's `fsync` depends on the persistence of C's data. Unfortunately (for schedulers), metadata sharing is common in file systems.

The inability to reorder is especially problematic for a deadline scheduler: a block-request deadline completely loses its relevance if one request's completion depends on the completion of unrelated I/Os. To demonstrate, we run two threads A (small) and B (big) with Linux's Block-Deadline scheduler [4], setting the block-request deadline to 20 ms for each. Thread A does 4 KB appends, calling `fsync` after each. Thread B does N bytes of random writes (N ranges from 16 KB to 4 MB) followed by an `fsync`. Figure 2.6 shows that even though A only writes one block each time, A's `fsync` latency depends on how much data B flushes each time.

Most file systems enforce ordering for correctness, so these problems occur with other crash-consistency mechanisms as well. For example, in

log-structured files systems [98], writes appended earlier are durable earlier.

2.2.2.3 Caching and Write Amplification

Sequentially reading or writing N bytes from or to a file often does not result in N bytes of sequential disk I/O for several reasons. First, file systems use different disk layouts, and layouts change as file systems age; hence, sequential file-system I/O may become random disk I/O. Second, file-system reads and writes may be absorbed by caches or write buffers without causing I/O. Third, some file-system features amplify I/O. For example, reading a file block may involve additional metadata reads, and writing a file block may involve additional journal writes. These behaviors prevent system-call schedulers from accurately estimating costs.

To show how this inability hurts system-call schedulers, we evaluate SCS-Token [36]. In SCS-Token, a process's resource usage is limited by the number of tokens it possesses. Per-process tokens are generated at a fixed rate, based on user settings. When the process issues a system call, SCS blocks the call until the process has enough tokens to pay for it.

We attempt to isolate a process A 's I/O performance from a process B by throttling B 's resource usage. If SCS-Token works correctly, A 's performance will vary little with respect to B 's I/O patterns. To test this behavior, we configure A to sequentially read from a large file while B runs workloads with different I/O patterns. Each of the B workloads involve repeatedly accessing R bytes sequentially from a 10 GB file and then randomly seeking to a new offset. We explore 7 values for R (from 4 KB to 16 MB) for both reads and writes (14 workloads total). In each case, B is throttled to 10 MB/s.

Figure 2.7 shows how A 's performance varies with B 's I/O patterns. Note the large gap between the performance of A with B reading vs. writing. When B is performing sequential writes, A 's throughput is as high as

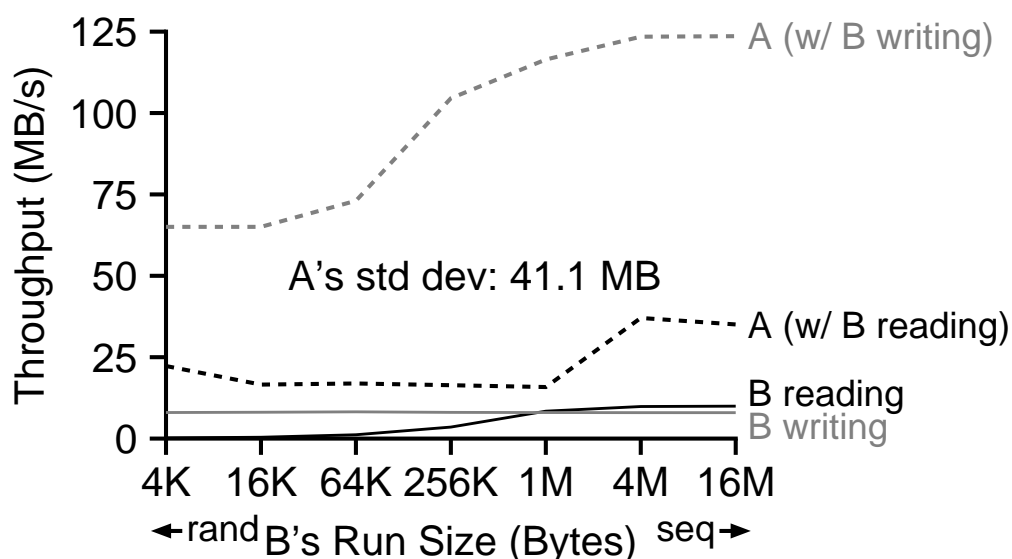


Figure 2.7: **SCS Token Bucket: Isolation.** *The performance of two processes is shown: a sequential reader, A, and a throttled process, B. B may read (black) or write (gray), and performs runs of different sizes (x-axis).*

125 MB/s; when B is performing random reads, A's throughput drops to 25 MB/s in the worst case. Writes appear cheaper than reads because write buffers absorb I/O and make it more sequential. Across the 14 workloads, A's throughput has a standard deviation of 41 MB, indicating A is highly sensitive to B's patterns. SCS-Token fails to isolate A's performance by throttling B, as SCS-Token cannot correctly estimate the cost of B's I/O pattern.

2.2.2.4 Discussion

Table 2.2 summarizes how different needs are met (or not) by each framework. The block-level framework fails to support correct cause mapping (due to write delegation such as journaling and delayed allocation) or control over reordering (due to file-system ordering requirements). The system-call framework solves these two problems, but fails to provide

	Block Syscall Split		
Cause Mapping	✘	✓	✓
Cost Estimation	✓	✘	✓
Reordering	✘	✓	✓

Table 2.2: **Framework Properties.** A ✓ indicates a given scheduling functionality can be supported with the framework, and an ✘ indicates a functionality cannot be supported.

enough information to schedulers for accurate cost estimation because it lacks low-level knowledge. These problems are general to many file systems; even if journals are not used, similar issues arise from the ordering constraints imposed by other mechanisms such as copy-on-write techniques [25] or soft updates [45]. Our split framework meets all the needs in Table 2.2 by incorporating ideas from the other two frameworks and exposing additional memory-related hooks.

2.3 Split Framework Design

Existing frameworks offer insufficient reordering control and accounting knowledge. Requests are queued, batched, and processed at many layers of the stack, thus the limitations of single-layer frameworks are unsurprising. We propose a holistic alternative: all important decisions about when to perform I/O work should be exposed as scheduling hooks, regardless of the level at which those decisions are made in the stack. We now discuss how these hooks support correct cause mapping (§2.3.1), accurate cost estimation (§2.3.2), and reordering (§2.3.3).

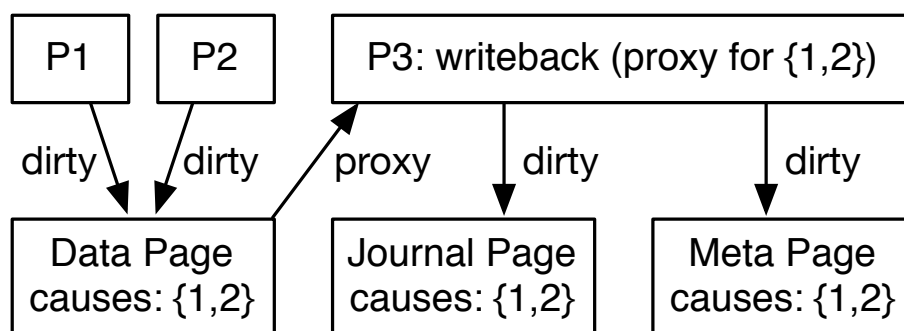


Figure 2.8: **Set Tags and I/O Proxies.** Our tags map metadata and journal I/O to the real causes, P1 and P2, not P3.

2.3.1 Cause Mapping

A scheduler must be able to map I/O back to the processes that caused it to accurately perform accounting even when some other process is submitting the I/O. Metadata is usually shared, and I/Os are usually batched, so there may be multiple causes for a single dirty page or a single request. Thus, the split framework tags I/O operations with sets of causes, instead of simple scalar tags (*e.g.*, those implemented by Mesnier *et al.* [88]).

Write delegation (§2.2.2.1) further complicates cause mapping when one process is dirtying data (not just submitting I/O) on behalf of other processes. We call such processes *proxies*; examples include the writeback and journaling tasks. Our framework tags proxy process to identify the set of processes being served instead of the proxy itself. These tags are created when a process starts dirtying data for others and cleared when it is done.

Figure 2.8 illustrates how our framework tracks multiple causes and proxies. Processes P1 and P2 both dirty the same data page, so the page’s tag includes both processes in its set. Later, a writeback process, P3, writes the dirty buffer to disk. In doing so, P3 may need to dirty the journal and metadata, and will be marked as a proxy for {P1, P2} (the tag is inherited from the page it is writing back). Thus, P1 and P2 are

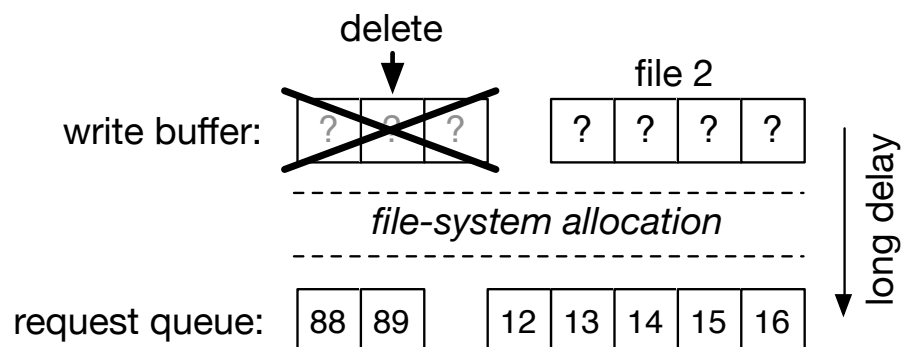


Figure 2.9: **Accounting: Memory vs. Block Level.** *Disk locations for buffered writes may not be known (indicated by the question marks on the blocks) if allocations are delayed.*

considered responsible when P3 dirties other pages, and the tag of these pages will be marked as such. The tag of P3 is cleared when it finishes submitting the data page to the block level.

2.3.2 Cost Estimation

Many policies require schedulers to know how much I/O costs, in terms of device time or other metrics. An I/O pattern's cost is influenced by file-system features, such as caches and write buffers, and by device properties (*e.g.*, random I/O is cheaper on flash than hard disk).

Costs can be most accurately estimated at the lowest levels of the stack, immediately above hardware (or better still in hardware, if possible). At the block level, request locations are known, so sequentiality-based models can estimate costs. Furthermore, this level is below all file-system features, so accounting is less likely to overestimate costs (*e.g.*, by counting cache reads) or underestimate costs (*e.g.*, by missing journal writes).

Unfortunately, writes may be buffered for a long time (*e.g.*, 30 seconds) before being flushed to the block level. Thus, while block-level accounting may accurately estimate the cost of a write, it is not aware of most

writes until some time after they enter the system via a `write` system call. Thus, if prompt accounting is more important than accurate accounting (*e.g.*, for interactive systems), accounting should be done at the memory level. Without memory-level information, a scheduler could allow a low-priority process to fill the write buffers with gigabytes of random writes, as we saw earlier (Figure 2.3).

Figure 2.9 shows the trade-off between accounting at the memory level (write buffer) and block level (request queue). At the memory level, schedulers do not know whether dirty data will be deleted before a flush, whether other writers will overwrite dirty data, or whether I/O will be sequential or random. A scheduler can guess how sequential buffered writes will be based on file offsets, but delayed allocation prevents certainty about the layout. After a long delay, on-disk locations and other details are known for certain at the block level.

The cost of buffered writes depends on future workload behavior, which is usually unknown. Thus, we believe all scheduling frameworks are fundamentally limited and cannot provide cost estimation that is both prompt and accurate. Our framework exposes hooks at both the memory and block levels, enabling each scheduler to handle the trade-off in the manner most suitable to its goals. Schedulers may even utilize hooks at both levels. For example, Split-Token (§2.5.3) promptly guesses write costs as soon as buffers are dirtied, but later revises that estimate when more information becomes available (*e.g.*, when the dirty data is flushed to disk).

2.3.3 Reordering

Most schedulers will want to reorder I/O to achieve good performance as well as to meet more specific goals (*e.g.*, low latency or fairness). Reordering for performance requires knowledge of the device (*e.g.*, whether it is useful to reorder for sequentiality), and is best done at a lower level

in the stack. We enable reordering at the block level by exposing hooks for both block reads and writes.

Unfortunately, the ability to reorder writes at the block level is greatly limited by file systems (§2.2.2.2). Thus, reordering hooks for writes (but not reads, which are not entangled by journals) are also exposed above the file system, at the system-call level. By controlling when write system calls run, a scheduler can control when writes become visible to the file system and prevent ordering requirements that conflict with scheduling goals.

Many storage systems have calls that modify metadata, such as `mkdir` and `creat` in Linux; the split framework also exposes these. This approach presents an advantage over the SCS framework, which cannot correctly schedule these calls. In particular, the cost of a metadata update greatly depends on file-system internals, of which SCS schedulers are unaware. Split schedulers, however, can observe metadata writes at the block level and accordingly charge the responsible applications.

File-system synchronization points (*e.g.*, `fsync` or similar) require all dependent data to be flushed to disk and typically invoke the file system's ordering mechanism. Unfortunately, logically independent operations often must wait for the synchronized updates to complete (§2.2.2.2), so the ability to schedule `fsync` is essential. Furthermore, writes followed by `fsync` are more costly than writes by themselves, so schedulers should be able to treat the two patterns differently. Thus, the split framework also exposes `fsync` scheduling.

2.4 Split Scheduling in Linux

Split-style scheduling could be implemented in a variety of storage stacks. In this work, we implement it in Linux, integrating with the ext4 and XFS file systems.

2.4.1 Cross-Layer Tagging

In Linux, I/O work is described by different function calls and data structures at different layers. For example, a write request may be represented by (a) the arguments to `vfs_write` at the system-call level, (b) a `buffer_head` structure in memory, and (c) a `bio` structure at the block level. Schedulers in our framework see the same requests in different forms, so it is useful to have a uniform way to describe I/O across layers. We add a causes tagging structure that follows writes through the stack and identifies the original processes that caused an I/O operation. Split schedulers can thereby correctly map requests back to the application from any layer.

Writeback and journal tasks are marked as I/O proxies, as described earlier (§2.3.1). In `ext4`, writeback calls the `ext4_da_writepages` function (“da” stands for “delayed allocation”), which writes back a range of pages of a given file. We modify this function so that as it does allocation for the pages, it sets the writeback thread’s proxy state as appropriate. For the journal proxy, we modify `jbd2` (`ext4`’s journal) to keep track of all tasks responsible for adding changes to the current transaction.

2.4.2 Scheduling Hooks

We now describe the hooks we expose, which are split across the system-call, memory, and block levels. Table 2.3 lists the split hooks.

System Call: These hooks allow schedulers to intercept entry and return points for various I/O system calls. A scheduler can delay the execution of a system call by simply sleeping in the entry hook. Like SCS, we intercept writes, so schedulers can separate writes before the file system entangles them. Unlike SCS, we do not intercept reads (no file-system mechanism entangles reads, so scheduling reads below the cache is preferable). Two metadata-write calls, `creat` and `mkdir`, and the Linux synchronization call, `fsync`, are also exposed to the scheduler. It would be useful

	Split Hook	Origin
System Call	<code>write-entry(syscall, ...)</code>	SCS
	<code>write-return(rv, ...)</code>	SCS
	<code>fsync-entry(syscall, ...)</code>	<i>new</i>
	<code>fsync-return(rv, ...)</code>	<i>new</i>
	<code>creat-entry(syscall, ...)</code>	<i>new</i>
	<code>creat-return(rv, ...)</code>	<i>new</i>
	<code>mkdir-entry(sys call, ...)</code>	<i>new</i>
	<code>mkdir-return(rv, ...)</code>	<i>new</i>
Mem	<code>buffer-dirty(oldset, new, ...)</code>	<i>new</i>
	<code>buffer-free(set)</code>	<i>new</i>
Block	<code>req-add(req, ...)</code>	block
	<code>req-complete(req, ...)</code>	block
	<code>req-merge(...)</code>	block
	<code>req-activate(req, ...)</code>	block
	<code>req-deactivate(req, ...)</code>	block
	<code>dispatch(...)</code>	block

Table 2.3: **Split Hooks.** The “Origin” column shows which hooks are new and which are borrowed from other frameworks.

(and straightforward) to support other metadata calls in the future (*e.g.*, `unlink`).

Note that in our implementation, the caller is blocked until the system call is scheduled. Other implementations are possible, such as buffering the system calls and returning immediately, or simply returning `EAGAIN` to tell the caller to issue the system call later. We choose this particular implementation because of its simplicity and POSIX compliance. Linux it-

self blocks writes (when there are too many dirty pages) and `fsyncs`, and most applications already deal with this behavior using separate threads; what we do is no different.

Memory: These hooks expose page-cache internals to schedulers. In Linux, a writeback thread (`pdflush`) decides when to pass I/O to the block-level scheduler, which then decides when to pass that I/O to disk. Both components are performing scheduling tasks, and separating them is inefficient (*e.g.*, writeback could flush more aggressively if it knew when the disk was idle). We add two hooks to inform the scheduler when buffers are dirtied or deleted. The `buffer-dirty` hook notifies the scheduler when a process dirties a buffer or when a dirty buffer is modified. In the latter case, the framework tells the scheduler which processes previously dirtied the buffer; depending on policy, the scheduler could revise accounting statistics, shifting some (or all) of the responsibility for the I/O to the last writer. The `buffer-free` hooks tell the scheduler if a buffer is deleted before writeback. Schedulers can either rely on Linux to perform writeback and throttle `write` system calls to control how much dirty data accumulates before writeback, or they can take complete control of the writeback. We evaluate the trade-off of these two approaches later (§2.7.1.2).

Block: These hooks are identical to those in Linux's original scheduling framework; schedulers are notified when requests are added to the block level or completed by the disk. Although we did not modify the function interfaces at this level, schedulers implementing these hooks in our framework are more informed, given tags within the request structures that identify the responsible processes. The Linux scheduling framework has over a dozen other block-level hooks for initialization, request merging, and convenience. We support all these as well for compatibility, but do not discuss them here.

```
def req_add(req):
    # put block-level read requests in a per-process queue
    if req.type == read:
        process = req.tag.get_cause()
        q = get_queue(process)
        enqueue(q, req)
    # immediately dispatch block-level write requests
    if req.type == write:
        send_to_disk(req)

def write_entry(syscall):
    process = syscall.tag.get_cause()
    q = get_queue(process)
    enqueue(q, syscall)
    sleep_until_scheduled(syscall)
    # after write_entry returns
    # the write system-call proceeds as usual
    return

def dispatch():
    process = get_min_cost_process()
    # task could be a block-level read request
    # or a system-call level syscall
    task = dequeue(get_queue(process))
    if task.type == read:
        send_to_disk(task)
    if task.type == write:
        signal_scheduled(task)

def req_complete(req):
    # accounting done at block level for both reads and writes
    disk_time = get_cost(req)
    process = req.tag.get_cause()
    add_cost(process, disk_time)
```

Figure 2.10: **Sample Scheduler Code.** Pseudo-code of a simplified split-level scheduler that implements fair scheduling.

2.4.3 Implement a Split-Level Scheduler

To demonstrate how to use the split hooks to implement a scheduling algorithm, Figure 2.10 lists the pseudo-code of a simplified split-level scheduler that implements fair scheduling (real schedulers would be implemented in C).

This scheduler employs a two-level scheduling strategy. Reads are scheduled at the block level to allow cache hits, while writes are scheduled at the system-call level to avoid any file-system entanglement. Whenever the scheduler receives a block-level read request (invokes `req-add`) or a syscall-level write call (invokes `write-entry`), it first finds the process that is responsible for the received I/O task by consulting the tag, then inserts the task to the queue corresponding to the process. Block-level writes received by the scheduler (also invokes `req-add`) are immediately dispatched to disk without any scheduling because writes are already subjected to scheduling at the higher system-call level.

Whenever the disk is idle, the `dispatch` hook is invoked to request more I/O tasks from the scheduler; the scheduler then checks which process has incurred the least I/O cost yet, and schedules I/O tasks from that process queue to ensure fair share between processes.

Accounting of the I/O cost (done within the `req-complete` hook) is performed at the block-level, regardless of the request type (read or write), because the block-level has the most knowledge of the location of the request and other information that determines the I/O cost.

2.4.4 Implementation Effort and Overhead

Implementing the split-level framework in Linux involves ~300 lines of code, plus some file-system integration effort, which we discuss later (§2.6). While representing a small change in the Linux code base, it enables powerful scheduling capabilities, as we will show next.

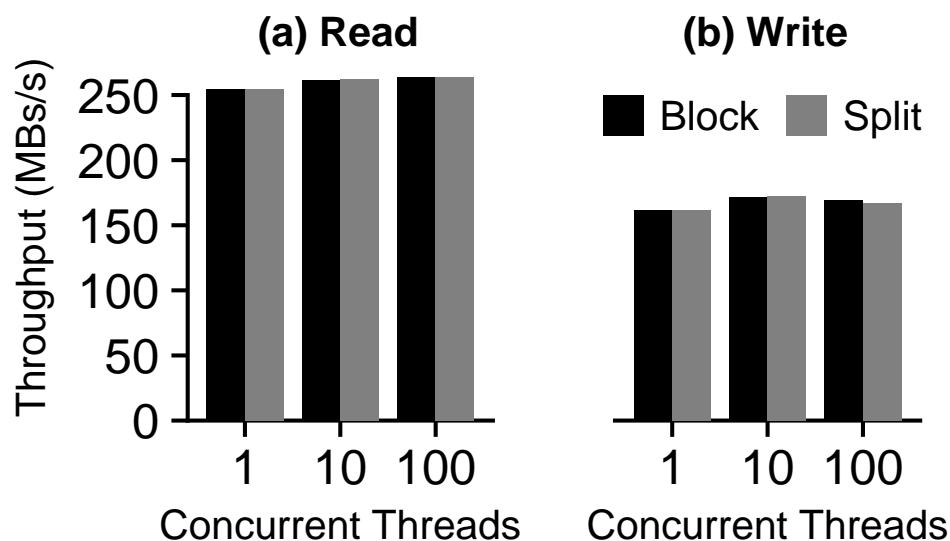


Figure 2.11: **Time Overhead.** *The split framework scales well with the number of concurrent threads doing I/O to an SSD.*

We now evaluate the time and space overhead of the split framework. In order to isolate framework overhead from individual scheduler overhead, we compare no-op schedulers implemented in both our framework and the block framework (a no-op scheduler issues all I/O immediately, without any reordering). Figure 2.11 shows our framework imposes no noticeable time overhead, even with 100 threads.

The split framework introduces some memory overhead for tagging writes with causes structures (§2.4.1). Memory overheads roughly correspond to the number of dirty write buffers. To measure this overhead, we instrument `kmalloc` and `kfree` to track the number of bytes allocated for tags over time. For our evaluation, we run HDFS with a write-heavy workload, measuring allocations on a single worker machine. Figure 2.12 shows the results: with the default Linux settings, average overhead is 14.5 MB (0.2% of total RAM); the maximum is 23.3 MB. Most tagging is on the write buffers; thus, a system tuned for more buffering should have

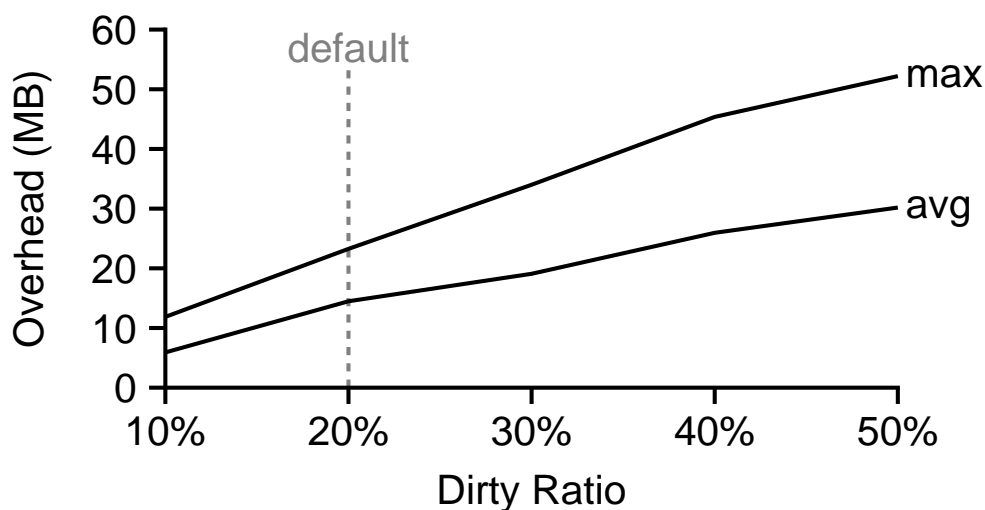


Figure 2.12: **Space Overhead.** *Memory overhead is shown for an HDFS worker with 8 GB of RAM under a write-heavy workload. Maximum and average overhead is measured as a function of the Linux `dirty_ratio` setting. `dirty_background_ratio` is set to half of `dirty_ratio`.*

higher tagging overheads. With a 50% `dirty_ratio` [6], maximum usage is still only 52.2 MB (0.6% of total RAM).

2.5 Scheduler Case Studies

In this section, we evaluate the split framework’s ability to support a variety of scheduling goals. We implement AFQ (§2.5.1), Split-Deadline (§2.5.2), and Split-Token (§2.5.3), and compare these schedulers to similar schedulers in other frameworks. Unless otherwise noted, all experiments run on top of ext4 with the Linux 3.2.51 kernel (most XFS results are similar but usually not shown). Our test machine has an eight-core, 1.4 GHz CPU and 16 GB of RAM. We use 500 GB Western Digital hard drives (AAKX) and an 80 GB Intel SSD (X25-M).

2.5.1 AFQ: Actually Fair Queuing

As shown earlier (§2.2.1), CFQ's inability to correctly map requests to processes causes unfairness, due to the lack of information Linux's elevator framework provides. Moreover, file-system ordering requirements limit CFQ's reordering options, causing priority inversions. In order to overcome these two drawbacks, we introduce AFQ (Actually Fair Queuing scheduler) to allocate I/O fairly among processes according to their priorities. More precisely, AFQ shares the disk head time among processes in proportion to $(8 - \text{prio})$, where prio is the priority of a process, ranging from 0 (high) to 7 (low).

Design: AFQ employs a two-level scheduling strategy. Reads are handled at the block level and writes (and calls that cause writes, such as `fsync`) are handled at the system-call level. This design allows reads to hit the cache while protecting writes from journal entanglement. Beneath the journal, low-priority blocks may be prerequisites for high-priority `fsync` calls, so writes at the block level are dispatched immediately.

AFQ chooses I/O requests to dequeue at the block and system-call levels using the stride algorithm [116]. Whenever a block request is dispatched to disk, AFQ charges the responsible processes for the disk I/O. The I/O cost is based on a simple seek model. In this model, the disk time needed to serve one request consists of the seek time and the transfer time, where the seek time is proportional to the offset difference between current request and the previous request, and the transfer time is proportional to the request size.

Evaluation: We compare AFQ to CFQ with four workloads, shown in Figure 2.13. Figure 2.13(a) shows read performance on AFQ and CFQ for eight threads, with priorities ranging from 0 (high) to 7 (low), each reading from its own file sequentially. We see that AFQ's performance is similar to CFQ, and both respect priorities.

Figure 2.13(b) shows asynchronous sequential-write performance, again

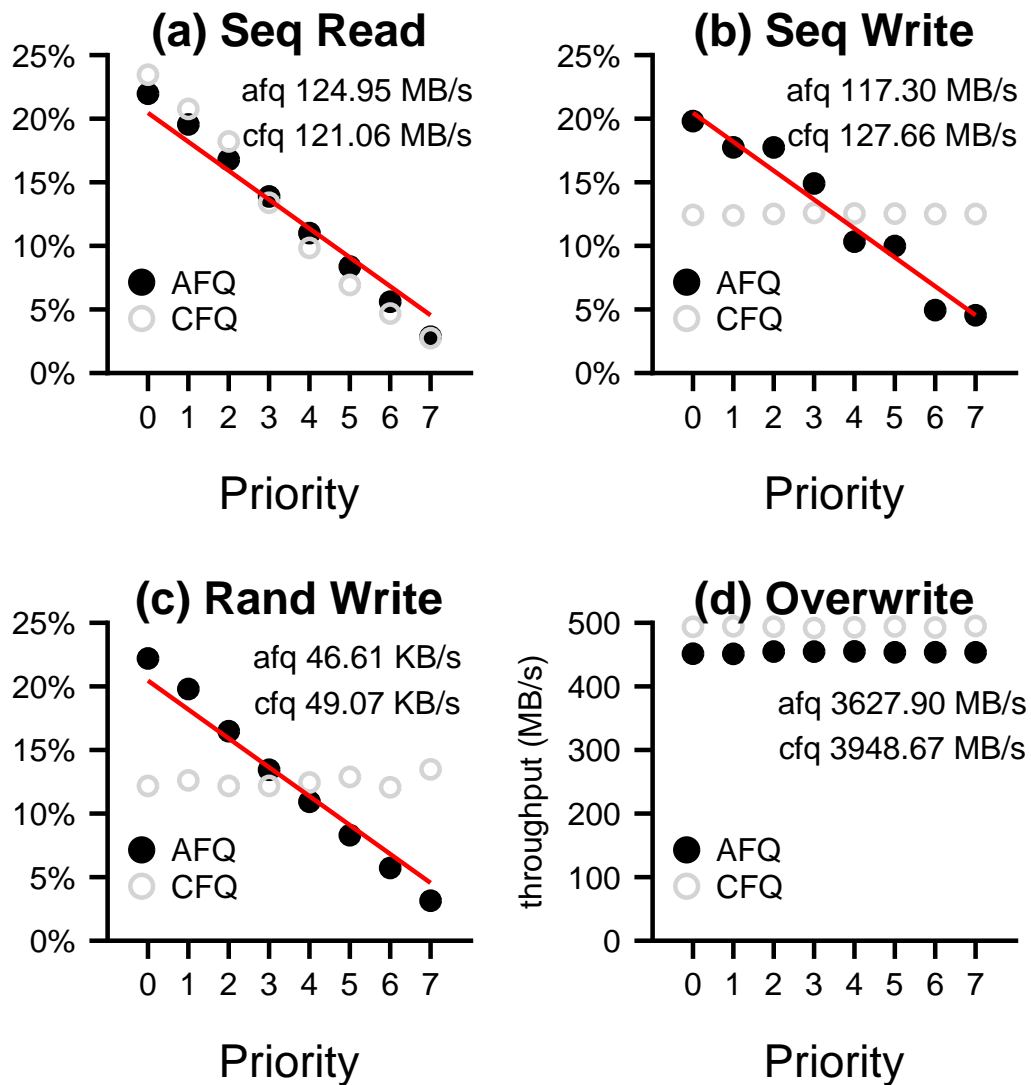


Figure 2.13: **AFQ Priority.** The plots show the percentage of throughput that threads of each priority receive. The lines show the goal distributions; the labels indicate total throughput.

with eight threads. This time, CFQ fails to respect priorities because of write delegation, but AFQ correctly maps I/O requests via split tags, and thus respects priorities. On average, CFQ deviates from the ideal by 82%, AFQ only by 16% (a 5× improvement).

Figure 2.13(c) shows synchronous random-write performance: we set up 5 threads per priority level, and each keeps randomly writing and flushing (with `fsync`) 4 KB blocks. The average throughput of threads at each priority level is shown. CFQ again fails to respect priority; using `fsync` to force data to disk invokes ext4's journaling mechanism and keeps CFQ from reordering to favor high-priority I/O. AFQ, however, blocks low-priority `fsyncs` when needed, improving throughput for high-priority threads. As shown, AFQ is able to respect priority, deviating from the ideal value only by 3% on average while CFQ deviates by 86% (a 28× improvement).

Finally, Figure 2.13(d) shows throughput for a memory-intense workload that just overwrites dirty blocks in the write buffer. One thread at each priority level keeps overwriting 4 MB of data in its own file. Both CFQ and AFQ get very high performance as expected, though AFQ is slightly slower (AFQ needs to do significant bookkeeping for each write system call). The plot has no fairness goal line as there is no contention for disk resources.

In general, AFQ and CFQ have similar performance; however, AFQ always respects priorities, while CFQ only respects priorities for the read workloads.

2.5.2 Deadline

As shown earlier (Figure 2.6 in §2.2.2.2), Block-Deadline does poorly when trying to limit tail latencies, due to its inability to reorder block I/Os in the presence of file-system ordering requirements. Split-level scheduling,

with system-call scheduling capabilities and memory-state knowledge, is better suited to this task.

Design: We implement the Split-Deadline scheduler by modifying the Linux deadline scheduler (Block-Deadline). Block-Deadline maintains two deadline queues and two location queues (for both read and write requests) [4]. In Split-Deadline, an `fsync`-deadline queue is used instead of a block-write deadline queue. During operation, if no read request or `fsync` is going to expire, block-level read and write requests are issued from the location queues to maximize performance. If some read requests or `fsync` calls are expiring, they are issued before their deadlines.

Split-Deadline monitors how much data is dirtied for one file using the `buffer-dirty` hook and thereby estimates the cost of an `fsync`. If there is an `fsync` pending that may affect other processes by causing too much I/O, it will not be issued directly. Instead, the scheduler asks the kernel to launch asynchronous writeback of the file's dirty data and waits until the amount of dirty data drops to a point such that other deadlines would not be affected by issuing the `fsync`. Asynchronous writeback does not generate a file-system synchronization point and has no deadline, so other operations are not forced to wait.

Evaluation: We compare Split-Deadline to Block-Deadline for a database-like workload on both hard disk drive (HDD) and solid state drive (SSD). We set up two threads A (small) and B (big); thread A appends to a small file one block (4 KB) at a time and calls `fsync` (this mimics database log appends) while thread B writes 1024 blocks randomly to a large file and then calls `fsync` (this mimics database checkpointing).

The deadline settings are shown in Table 2.4. We choose shorter block-write deadlines than `fsync` deadlines because each `fsync` causes multiple block writes; however, our results do not appear sensitive to the exact values chosen. Linux's Block-Deadline scheduler does not support setting different deadlines for different processes, so we add this feature to enable

	A		B	
	Block Write	Fsync	Block Write	Fsync
HDD	10 ms	100 ms	100 ms	6000 ms
SSD	1 ms	3 ms	10 ms	100 ms

Table 2.4: **Deadline Settings.** For *Block-Deadline*, we set deadlines for block-level writes; for *Split-Deadline*, we set deadlines for *fsyncs*.

a fair comparison.

Figure 2.14 shows the experiment results on both HDD and SSD. We can see that when no I/O from B is interfering, both schedulers give A low-latency *fsyncs*. After B starts issuing big *fsyncs*, however, *Block-Deadline* starts to fail: A’s *fsync* latencies increase by an order of magnitude; this happens because B generates too much bursty I/O when calling *fsync*, and the scheduler has no knowledge of or control over when they are coming. Worse, A’s operations become dependent on these I/Os.

With *Split-Deadline*, however, A’s *fsync* latencies mostly fluctuate around the deadline, even when B is calling *fsync* after large writes. Sometimes A exceeds its goal slightly because our estimate of the *fsync* cost is not perfect, but latencies are always relatively near the target. Such performance isolation is possible because *Split-Deadline* can reorder to spread the cost of bursty I/Os caused by *fsync* without forcing others to wait.

2.5.3 Token Bucket

Earlier, we saw that *SCS-Token* [36] fails to isolate performance (Figure 2.7 in §2.2.2.3). In particular, the throughput of a process A was sensitive to the activities of another process B. *SCS* underestimates the I/O cost of some B workloads, and thus does not sufficiently throttle B. In this section, we evaluate *Split-Token*, a reimplementaion of token bucket in our framework that caps the I/O used by each throttled process to a pre-specified rate to guarantee isolation.

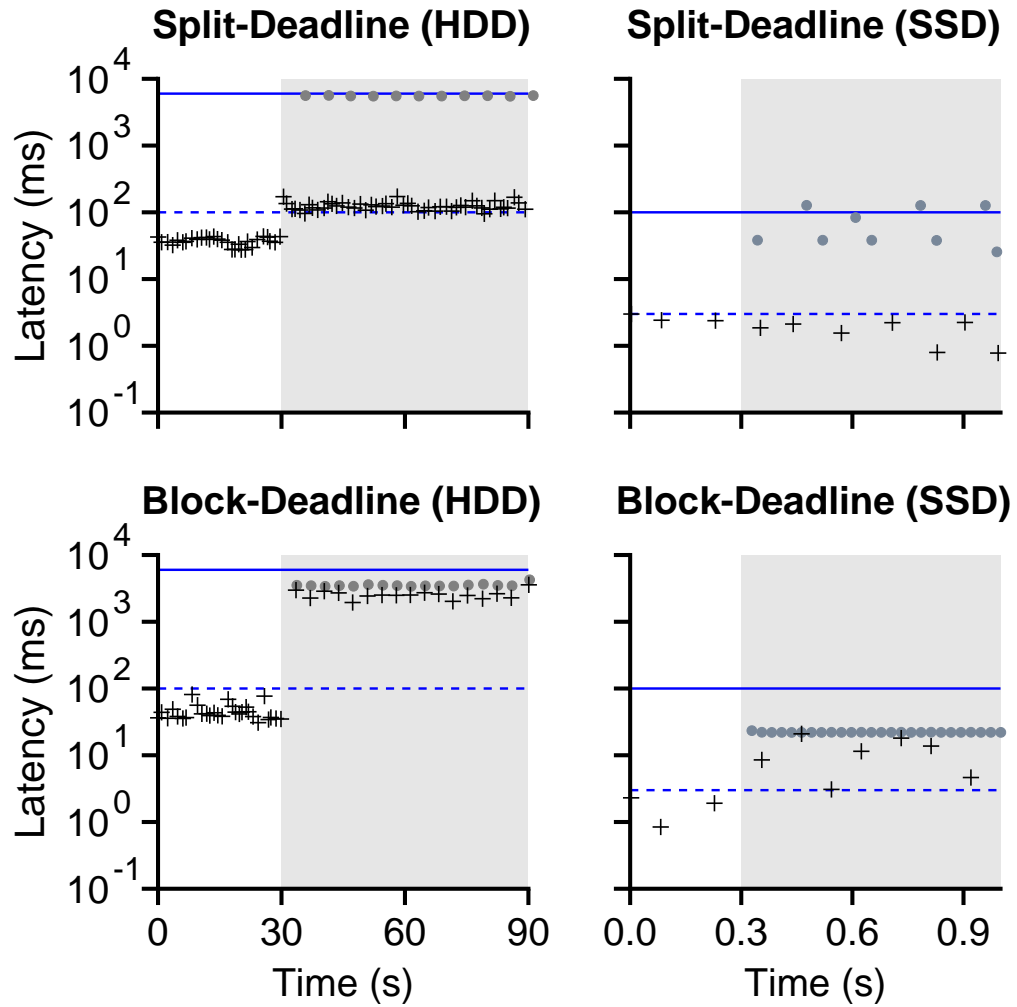


Figure 2.14: **Fsync Latency Isolation.** Dashed and solid lines present the goal latencies of A and B respectively. Dots represent the actual latency of B's calls, and pluses represent the actual latency of A's calls. The shaded area represents the time when B's fsyncs are being issued.

Design: As with SCS-Token, throttled processes are given tokens at a set rate. I/O costs tokens, I/O is blocked if there are no tokens, and the number of tokens that may be held is capped. Split-Token throttles a process's system-call writes and block-level reads if and only if the number of tokens is negative. System-call reads are never throttled (to utilize the cache). Block writes are never throttled (to avoid entanglement).

Our implementation uses memory-level and block-level hooks for accounting. The scheduler promptly charges tokens as soon as buffers are dirtied, and then revises when the writes are later flushed to the block level (§2.3.2), charging more tokens (or refunding them) based on amplification and sequentiality. Tokens represent bytes, so accounting normalizes the cost of an I/O pattern to the equivalent amount of sequential I/O (*e.g.*, 1 MB of random I/O may be counted as 10 MB).

Split-Token estimates I/O cost based on two models, both of which assume an underlying hard disk (simpler models could be used on SSD). When buffers are first dirtied at the memory level, a preliminary model estimates cost based on the randomness of request offsets within the file. Later, when the file system allocates space on disk for the requests and flushes them to the block level, a disk model (similar to the one we use in §2.5.1) revises the cost estimate. The second model is more accurate because it can consider more factors than the first model, such as whether the file system introduced any fragmentation, and whether the file is located near other files being written.

Evaluation: We repeat our earlier SCS experiments (Figure 2.7) with Split-Token, as shown in Figure 2.15. We observe that whether B does reads or writes has little effect on A (the A lines are near each other). Whether B's pattern is sequential or random also has little impact (the lines are flat). Across all workloads, the standard deviation of A's performance is 7 MB, about a 6× improvement over SCS (SCS-Token's deviation was 41 MB).

We now directly compare SCS-Token with Split-Token using a broader

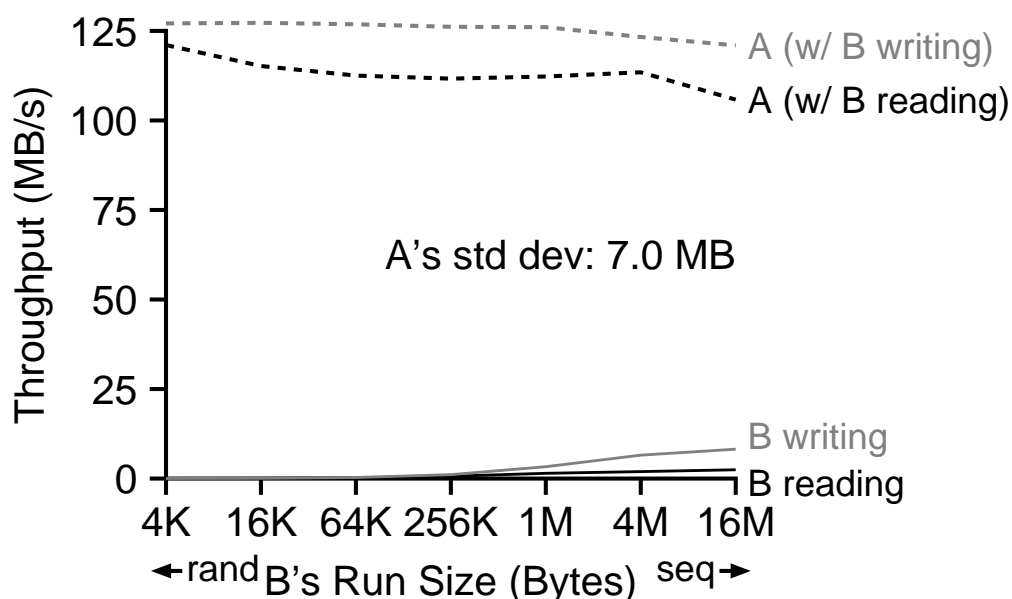


Figure 2.15: **Isolation: Split-Token with ext4.** The same as Figure 2.7, but for our Split-Token implementation. A is the unthrottled sequential reader, and B is the throttled process performing I/O of different run sizes.

range of read and write workloads for process B. I/O can be random (expensive), sequential, or served from memory (cheap). As before, A is an unthrottled reader, and B is throttled to 1 MB/s of normalized I/O. Figure 2.16 (left) shows that Split-Token is near the isolation target all six times, whereas SCS-Token significantly deviates three times (twice by more than 50%), again showing Split-Token provides better isolation.

After isolation, a secondary goal is the best performance for throttled processes, which we measure in Figure 2.16 (right). Sometimes B is faster with SCS-Token, but only because SCS-Token is incorrectly sacrificing isolation for A (e.g., B does faster random reads with SCS-Token, but A's performance drops over 80%). We consider the cases where SCS-Token did provide isolation. First, Split-Token is $2.3\times$ faster for “read-mem”. SCS-Token logic must run on every read system call, whereas Split-Token does not. SCS-Token still achieves nearly 2 GB/s, though, indicating cache

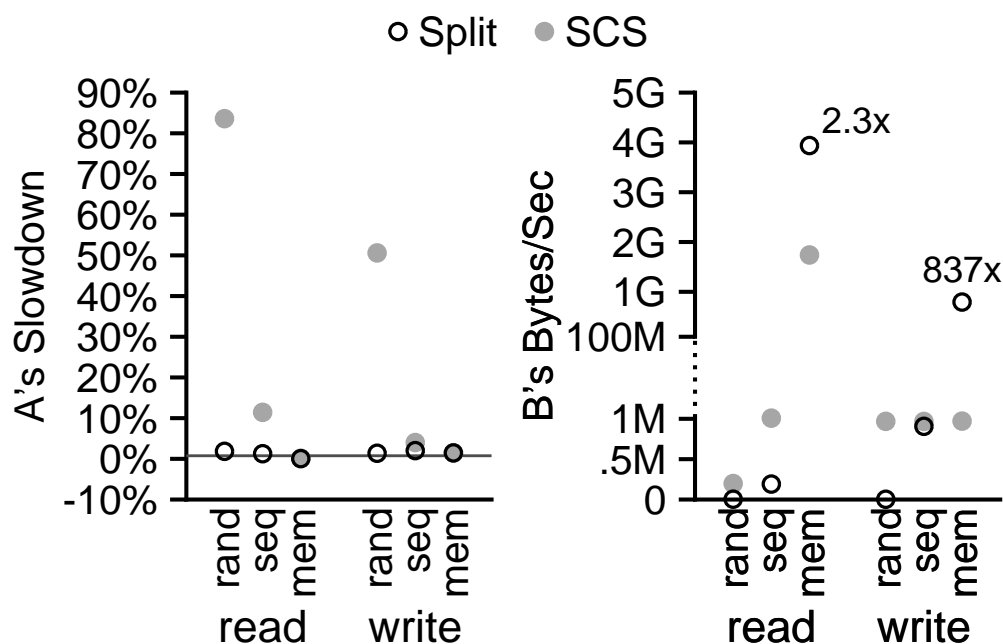


Figure 2.16: **Split-Token vs. SCS-Token.** *Left: A's throughput slowdown is shown. Right: B's performance is shown. Process A achieves about 138 MB/s when running alone, and B is throttled to 1 MB/s of normalized I/O, so there should be a 0.7% slowdown for A (shown by a target line). The x-axis indicates B's workload; A always reads sequentially.*

hits are not throttled. Although the goal of SCS-Token was to do system-call scheduling, Craciunas *et al.* needed to modify the file system to tell which reads are cache hits [37]. Second, Split-Token is 837 \times faster for “write-mem”. SCS-Token does write accounting at the system-call level, so it does not differentiate buffer overwrites from new writes. Thus, SCS-Token unnecessarily throttles B. With Split-Token, B's throughput does not reach 1 MB/s for “read-seq” because the intermingled I/Os from A and B are no longer sequential; we charge it to both A and B.

We finally evaluate Split-Token for a large number of threads; we repeat the six workloads of Figure 2.16, this time varying the number of B threads performing the I/O task (all threads of B share the same I/O

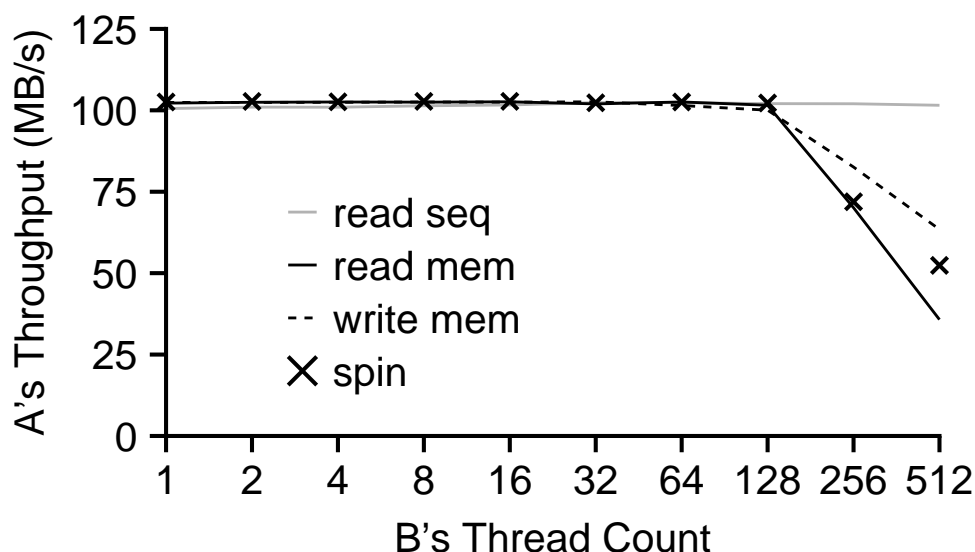


Figure 2.17: **Split-Token Scalability.** *A's throughput is shown as a function of the number of B threads performing a given activity. Goal performance is 101.7 MB (these numbers were taken on a 32-core CloudLab node with a 1 TB drive).*

limit). Figure 2.17 shows the results. For sequential read, the number of B threads has no impact on A's performance, as desired. We do not show random read, sequential write, or random write, as these lines would appear the same as the read-sequential line (varying at most by 1.7%). However, when B is reading or writing to memory, A's performance is only steady if B has 128 threads or less. Since the B threads do not incur any disk I/O, our I/O scheduler does not throttle them, leaving the B threads free to dominate the CPU, indirectly slowing A. To confirm this, we do an experiment (also shown in Figure 2.17) where B threads simply execute a spin loop, issuing no I/O; A's performance still suffers in this case. This reminds us of the usefulness of CPU schedulers in addition to I/O schedulers: if a process does not receive enough CPU time, it may not be able to issue requests fast enough to fully utilize the storage system.

2.5.4 Implementation Effort

Implementing different schedulers within the split framework is not only possible, but relatively easy as it only requires implementing relevant hooks (see §2.4.3). Split-AFQ takes ~950 lines of code to implement from scratch, Split-Deadline takes ~750 lines of code, and Split-Token takes ~950 lines of code. As a comparison, Block-CFQ takes more than 4000 lines of code (though it includes many performance optimizations that AFQ does not offer), Block-Deadline takes ~500 lines of code, and SCS-Token takes ~2000 lines of code (SCS-Token is large because there is not a clean separation between the scheduler and framework).

Additional complexity of split-level schedulers compared to their block-level counterparts comes from scheduling system calls to work around file system ordering requirements. For example, in addition to maintaining the deadline and location queues as Block-Deadline does, the Split-Deadline scheduler has to also track whether an `fsync` would prevent the scheduler from reordering requests to meet their deadlines, and take necessary measures to avoid the situation. We feel that this complexity is justified because it enables correct scheduling. However, if one does not wish to afford the additional complexity, Block-Deadline and other block-level schedulers can natively run within the split framework as we provide backward compatibility.

2.6 File System Integration

Thus far we have presented results with `ext4`; now, we consider the effort necessary to integrate `ext4` and other file systems, in particular `XFS`, into the split framework.

Integrating a file system involves (a) tagging relevant data structures the file system uses to represent I/O in memory and (b) identifying the proxy mechanisms in the file system and properly tagging the proxies.

In Linux, part (a) is mostly file-system independent as many file systems use generic page buffer data structures to represent I/O. Both ext4 and XFS rely heavily on the `buffer_head` structure, which we already tag properly. Thus we are able to integrate XFS buffers with split tags by adding just two lines of code, and ext4 with less than 10 lines. In contrast, btrfs [82] uses its own buffer structures, so integration would require more effort.

Part (b), on the other hand, is highly file-system specific, as different file systems use different proxy mechanisms. For ext4, the journal task acts as a proxy when writing the physical journal, and the writeback task acts as a proxy when doing delayed allocation. XFS uses logical journaling, and has its own journal implementation. For a copy-on-write file system, garbage collection would be another important proxy mechanism. Properly tagging these proxies is a bit more involved. In ext4, it takes 80 lines of code across 5 different files. Fortunately, such proxy mechanisms typically only involve metadata, so for data-dominated workloads, partial integration with only (a) should work relatively well.

In order to verify the above hypotheses, we have fully integrated ext4 with the split framework, and only partially integrated XFS with part (a). We evaluate the effectiveness of our partial XFS integration on both data-intensive and metadata-intensive workloads.

Figure 2.18 repeats our earlier isolation experiment (Figure 2.15), but with XFS; these experiments are data-intensive. Split-Token again provides significant isolation, with A only having a deviation of 12.8 MB. In fact, all the experiments we show earlier are data intensive, and XFS has similar results (not shown) as ext4.

Figure 2.19 shows the performance of a metadata-intense workload for both XFS and ext4. In this experiment, A reads sequentially while B repeatedly creates empty files and flushes them to disk with `fsync`. B is throttled, A is not. B sleeps between each create for a time varied on the



Figure 2.18: **Isolation: Split-Token with XFS.** This is the same as Figure 2.7 and Figure 2.16, but for XFS running with our Split implementation of token bucket.

x-axis. As shown in the left plot, B’s sleep time influences A’s performance significantly with XFS, but with ext4 A is isolated. The right plot explains why: with ext4, B’s creates are correctly throttled, regardless of how long B sleeps. With XFS, however, B is unthrottled because XFS does not give the scheduler enough information to map the metadata writes (which are performed by journal tasks) back to B.

We conclude that some file systems can be partially integrated with minimal effort, and data-intense workloads will be well supported. Support for metadata workloads, however, requires more effort.

2.7 Real Applications

In this section, we explore whether the split framework is a useful foundation for databases (§2.7.1), virtual machines (§2.7.2), and distributed file systems (§2.7.3).

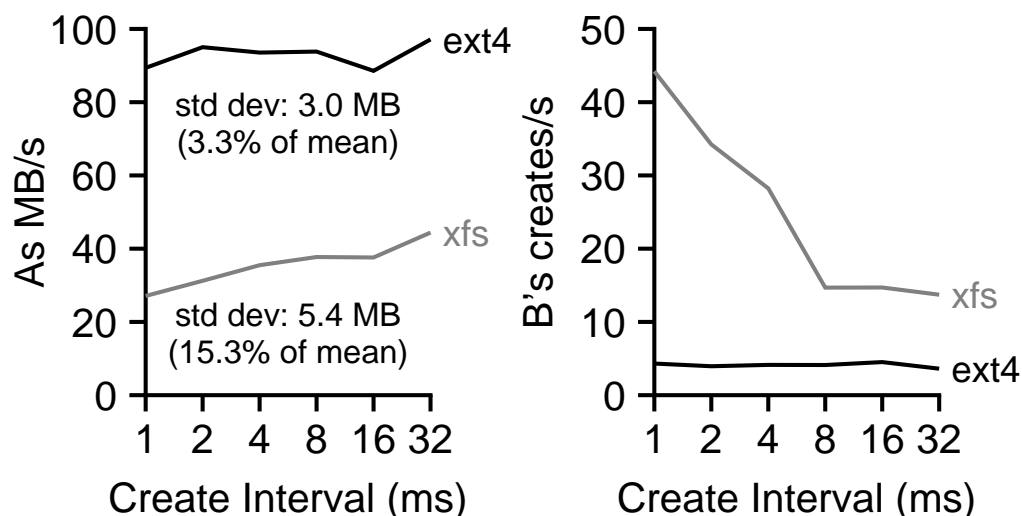


Figure 2.19: **Metadata: Split-Token with XFS and ext4.** Process A sequentially reads while B creates and flushes new, empty files. A’s throughput is shown as function of how long B sleeps between operations (left). B’s create frequency is also shown for the same experiments (right).

2.7.1 Databases

To show how real databases could benefit from Split-Deadline’s low-latency `fsyncs`, we measure transaction-response time for SQLite3 [61] and PostgreSQL [12] running with both Split-Deadline and Block-Deadline.

2.7.1.1 SQLite3

We run SQLite3 on a hard disk drive. For Split-Deadline, we set short deadlines (100 ms) for `fsyncs` on the write-ahead log file and reads from the database file and set long deadlines (10 seconds) for `fsyncs` on the database file. For Block-Deadline, the default settings (50 ms for block reads and 500 ms for block writes) are used. We make minor changes to SQLite to allow concurrent log appends and checkpointing and to set appropriate deadlines. For our benchmark, we randomly update rows in a large table, measure transaction latencies, and run checkpointing in a

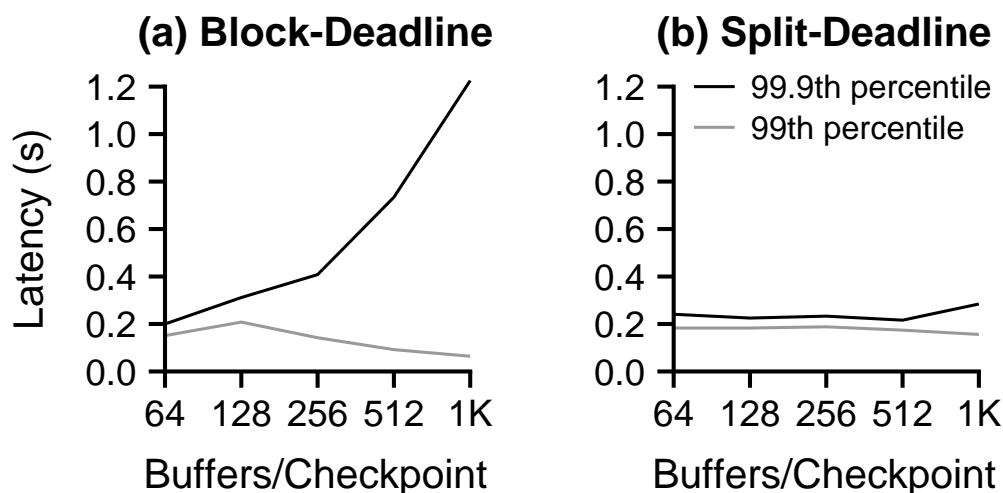


Figure 2.20: **SQLite Transaction Latencies.** *99th and 99.9th percentiles of the transaction latencies are shown. The x-axis is the number of dirty buffers we allow before checkpoint.*

separate thread whenever the number of dirty buffers reaches a threshold.

Figure 2.20(a) shows the transaction tail latencies (99th and 99.9th percentiles) when we change the checkpointing threshold. When checkpoint thresholds are larger, checkpointing is less frequent, fewer transactions are affected, and thus the 99th line falls. Unfortunately, this approach does not eliminate tail latencies; instead, it concentrates the cost on fewer transactions, so the 99.9th line continues to rise. In contrast, Figure 2.20(b) shows that Split-Deadline provides much smaller tail latencies (a 4× improvement for 1K buffers).

2.7.1.2 PostgreSQL

We run PostgreSQL [12] on top of an SSD and benchmark it using `pgbench` [5], a TPC-B like workload. We change PostgreSQL to set I/O deadlines for each worker thread. We want consistently low transaction latencies (within 15 ms), so we set the foreground `fsync` deadline to 5 ms,

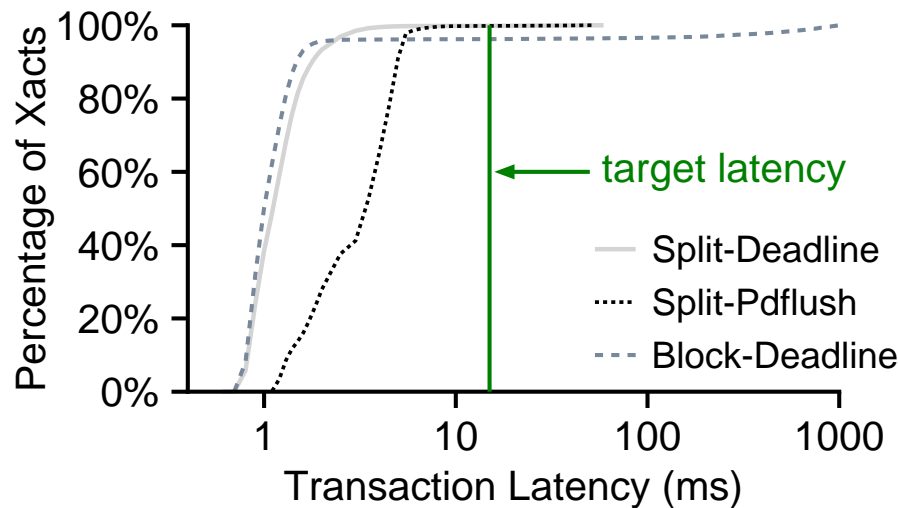


Figure 2.21: **PostgreSQL Transaction Latencies.** A CDF of transaction latencies is shown for three systems. Split-Pdflush is Split-Deadline, but with pdflush controlling writeback separately.

and the background checkpointing `fsync` deadline to 200 ms for Split-Deadline. For Block-Deadline, we set the block write deadline to 5 ms. For block reads, a deadline of 5 ms is used for both Split-Deadline and Block-Deadline. Checkpoints occur every 30 seconds.

Figure 2.21 shows the cumulative distribution of the transaction latencies. We can see that when running on top of Block-Deadline, 4% of transactions fail to meet their latency target, and over 1% take longer than 500 ms. After further inspection, we found that the latency spikes happen at the end of each checkpoint period, when the system begins to flush a large amount of dirty data to disk using `fsync`. Such data flushing interferes with foreground I/Os, causes long transaction latency and low system throughput. The database community has long experienced this “`fsync` freeze” problem, and has no great solution for it [3, 11, 12]. We show next that Split-Deadline provides a simple solution to this problem.

When running Split-Deadline, we have the ability to schedule `fsyncs`

and minimize their performance impact to foreground transactions. However, `pdflush` (Linux’s writeback task) may still submit many writeback I/Os without scheduler involvement and interfere with foreground I/Os. Split-Deadline maintains deadlines in this case by limiting the amount of data `pdflush` may flush at any given time by throttling write system calls. In Figure 2.21 we can see that this approach effectively eliminates tail latency: 99.99% of the transactions are completed within 15 ms. Unfortunately, the median transaction latency is much higher because write buffers are not fully utilized.

When `pdflush` is disabled, though, Split-Deadline has complete control of writeback, and can allow more dirty data in the system without worrying about untimely writeback I/Os. It then initiates writeback in a way that both observes deadlines and optimizes performance, thus eliminating tail latencies while maintaining low median latencies, as shown in Figure 2.21.

2.7.2 Virtual Machines (QEMU)

Isolation is especially important in cloud environments, where customers expect to be isolated from other (potentially malicious) customers. To evaluate our framework’s usefulness in this environment, we repeat our token-bucket experiment in Figure 2.16, this time running the unthrottled process A and throttled process B in separate QEMU instances. The guests run a vanilla kernel; the host runs our modified kernel. Thus, throttling is on the whole VM, not just the benchmark we run inside. We use an 8 GB machine with a four-core 2.5 GHz CPU.

Figure 2.22 shows the results for QEMU running over both SCS and Split-Token on the host. The isolation results for A (left) are similar to the results when we ran A and B directly on the host (Figure 2.16): with Split-Token, A is always well isolated, but with SCS, A experiences major slowdowns when B does random I/O.

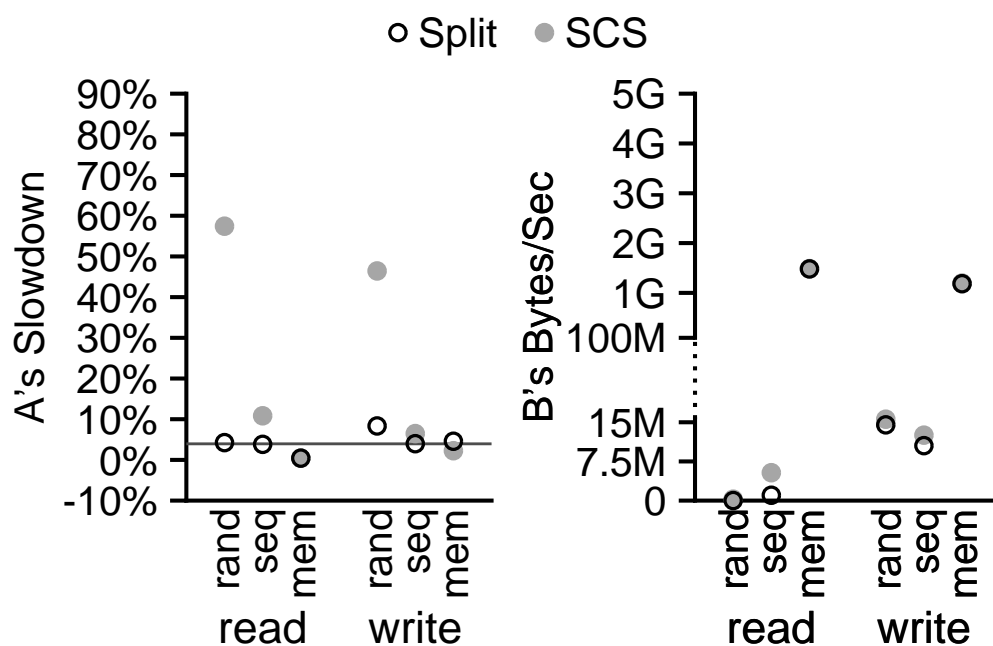


Figure 2.22: **QEMU Isolation.** This is the same as Figure 2.16, but processes A and B run in different QEMU virtual machines ext4 on the host. B is throttled to 5 MB/s. Reported throughput is for the processes at the guest system-call level.

The throughput results for B (right) are more interesting: whereas before SCS greatly slowed memory-bound workloads, now SCS and Split-Token provide equal performance for these workloads. This is because when a throttled process is memory bound, it is crucial for performance that a caching/buffering layer exist above the scheduling layer. The split and QEMU-over-SCS stacks have this property (and memory workloads are fast), but the raw-SCS stack does not.

2.7.3 Distributed File Systems (HDFS)

To show that local split scheduling is a useful foundation to provide isolation in a distributed environment, we integrate HDFS with Split-Token to provide isolation to HDFS clients. We modify the client-to-worker

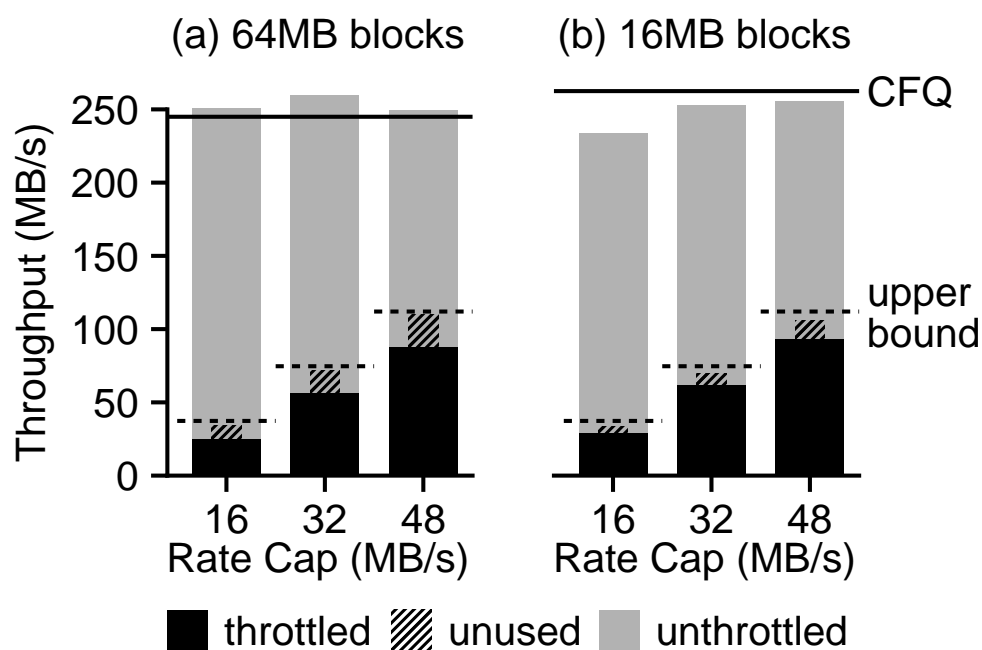


Figure 2.23: **HDFS Isolation.** *Solid-black and gray bars represent the total throughput of throttled and unthrottled HDFS writers, respectively. Dashed lines represent an upper bound on throughput; solid lines represent Block-CFQ throughput.*

protocol so workers know which account should be billed for disk I/O generated by the handling of a particular RPC call. Account information is propagated down to Split-Token and across to other workers (for pipelined writes).

We evaluate our modified HDFS on a 256-core CloudLab cluster (one NameNode and seven workers, each with 32 cores). Each worker has 8 GB of RAM and a 1 TB disk. We run an unthrottled group of four threads and a throttled group of four threads. Each thread sequentially writes to its own HDFS file.

Figure 2.23(a) shows the result for varying rate limits on the x-axis. The summed throughput (*i.e.*, that of both throttled and unthrottled writers) is similar to throughput when HDFS runs over CFQ without any pri-

orities set. With Split-Token, though, smaller rate caps on the throttled threads provide the unthrottled threads with better performance (*e.g.*, the gray bars get more throughput when the black bars are locally throttled to 16 MB/s).

Given there are seven datanodes, and data must be triply written for replication, the expected upper bound on total I/O is $(\text{ratecap}/3)*7$. The dashed lines show these upper bounds in Figure 2.23(a); the black bars fall short. We found that many tokens go unused on some workers due to load imbalance. The hashed black bars represent the potential HDFS write I/O that was thus lost.

In Figure 2.23(b), we try to improve load balance by decreasing the HDFS block size from 64 MB (the default) to 16 MB. With smaller blocks, fewer tokens go unused, and the throttled writers achieve I/O rates nearer the upper bound. We conclude that local scheduling can be used to meet distributed isolation goals; however, throttled applications may get worse-than-expected performance if the system is not well balanced.

2.8 Conclusion

In this work, we have shown that single-layer schedulers operating at either the block level or system-call level fail to support common goals due to a lack of coordination with other layers.

While our experiments indicate that simple layering must be abandoned, we need not sacrifice modularity. In our split framework, the scheduler operates across all layers, but is still abstracted behind a collection of handlers. This approach is relatively clean, and enables pluggable scheduling. Supporting a new scheduling goal simply involves writing a new scheduler plug-in, not re-engineering the entire storage system.

Our hope is that split-level scheduling will inspire future vertical integration in storage stacks. The source code of the split framework and

three individual split-level schedulers can be found at
<http://research.cs.wisc.edu/adsl/Software/split>.

3

Thread Architecture Diagrams

In Chapter 2 we focused on the scheduling in local storage stacks; we now move to the scheduling in distributed storage systems. Local scheduling forms the foundation for any effective scheduling in distributed systems. However, distributed scheduling has its own set of challenges. In particular, modern storage systems are complex, concurrent programs; many systems are realized via an intricate series of stages, queues, and thread pools, based loosely on the SEDA design principle [120]. Understanding how to introduce scheduling control into these systems is challenging; a single request may flow through numerous stages across multiple machines before its completion.

Due to these challenges, effective scheduling in distributed storage systems has remained difficult despite repeated attempts from both industry and academia. Existing systems usually provide weak or no performance guarantees as they lack the scheduling support to realize such guarantees. For example, HBase [46] places the scheduling control at the entry point of RPC handling; however, as we show later (§4.4), the RPC handling threads in HBase interact with other threads in unexpected ways, causing the schedulers placed here to fail. Similarly, Cassandra [73] used to schedule requests immediately after they enter the system, but only finds that it is insufficient to achieve proper scheduling and isolation: scheduling at this point is ineffective due to Cassandra's complex internal structure [15].

To overcome the challenges, we demonstrate a method to discover the *schedulability* of concurrent storage systems. Our method traces a system of interest under various workloads and produces a *Thread Architecture Diagram (TAD)*. TAD models the thread behavior and interactions of a system; by analyzing a TAD, scheduling problems of the said system can be discerned, pointing towards solutions that introduce necessary scheduling controls. TADs can be used to model and analyze any multiple-threaded programs as it reveals how different threads interact with each other and consume resources; it is particularly useful in analyzing distributed storage systems, where a request could flow through many stages across multiple nodes, consuming various types of resources (I/O, network, CPU, etc.) along its path.

In this chapter, we first introduce how we model the system behaviors using TAD, and the general notations used (§3.1). We then discuss *TADalyzer*, a tool we develop to automatically produce TADs for various systems based on dynamic instrumentation and tracing techniques (§3.2). Next we produce TADs for four important and widely-used scalable storage systems and discuss their behaviors under the lens of TAD. We discuss HBase/HDFS [46, 104] in §3.3, MongoDB [34] in §3.4, Cassandra [73] in §3.5, and Riak KV [71] in §3.6. Finally, we conclude (§3.7).

3.1 Thread Architecture Diagrams

At the highest level, 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, isolation, latency guarantees, or other SLOs). This plan is distributed to and enforced by local schedulers at different points along the data path, as shown in Figure 3.1.








	Bounded stage [Boxes above: the resources it uses (square brackets means “may or may not use”). Boxes below: its scheduling problems. ‘(#): Default number of threads.]
	On-demand stage.
	Scheduling point [Plug: allows pluggable schedulers. Lock: ordering constrains.]
A → B	Data flow [Stage A issues requests to stage B]
A - - - - -> B	Blocking relationship [Stage A blocks on the stage B]
A —◇—◇—◇B	Information Flow
	Process boundary (stages within the same process share address space)
	Node boundary
	CPU, I/O, network, lock resource [Left to Right]
	No-scheduling, Unknown resource usage, hidden-Contention, Blocking, Ordering-constraint problem [Left to Right]

Table 3.1: Notation for Thread Architecture Diagrams.

How well scheduling policies can be realized depends on specific features of the data path. In modern SEDA-based distributed storage systems, the data path consists of many distinct stages that each exhibit complex behaviors. We introduce thread architecture diagrams to visualize these behaviors. Table 3.1 summarizes the building blocks used in TADs.

A TAD models a system as a collection of *nodes*, each of which consists of *stages*. Arrows indicate the flow of requests between nodes and stages. *Resources* are consumed within a stage as requests are processed; in a TAD, the resources shown as consumed include CPU, I/O, network, and locks (with four different symbols).

A stage contains threads performing similar tasks (e.g., handling RPC

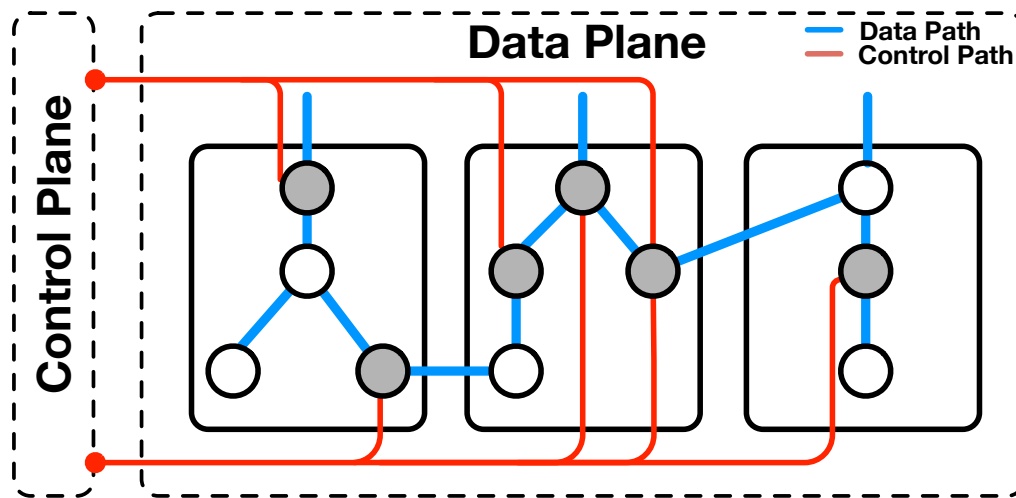


Figure 3.1: **Scheduling Framework Model.** Circles represent stages along the data path (gray indicates scheduling within the stage)

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 implemented by a virtual machine) regardless of its implementation. Within a stage, threads can be organized as a thread pool with a fixed (or maximum) number of active threads (*bounded stage*) or can be allocated dynamically as new requests arrive (*on-demand stage*). In a TAD, on-demand stages are shaded in gray.

Each bounded stage has an associated queue from which threads pick tasks; each queue is a potential *scheduling point* where scheduling policies can be realized by reordering requests. The queue can be either implicit (e.g., the default FIFO queue associated with a Java thread pool) or explicit (with an API to allow choice of policy, or hard-coded decisions). Even with an explicit queue, some requests may need to be served in order for correctness; this is an *ordering constraint*. In a TAD, both explicit and implicit queues are shown; the plug symbol designates the scheduler may be configured with different policies and the lock symbol indicates an ordering constraint.

After a thread issues a request to downstream stages, the thread may immediately proceed to other requests (*asynchronous*), or block until notified that the request is completed at the downstream stage (*synchronous*). Blocking is indicated in a TAD with a dashed line between the downstream stage and the blocked stage.

3.2 TADalyzer

TADs are automatically obtained using TADalyzer, a tool we developed to automatically discover thread architecture using instrumentation and tracing techniques. TADalyzer requires the user to annotate the code to identify important stages; it then tracks thread activities (i.e., creation, termination, resource consumption, blocking on signals) and interactions to construct the data flow and interactions between stages. For now TADalyzer only works for storage systems based on Java or C++, but it can be extended to support other languages as well.

3.2.1 Automatic Discovery

To discover the thread architecture of a system, the user first needs to annotate the code to identify important stages in the system. TADalyzer then uses binary instrumentation tools (byteman [41] for Java and Pin [76] for C++) to monitor thread creation and termination, and builds a mapping between threads and stages based on the annotation. Any threads that are not explicitly annotated map to a special NULL stage. Using this mapping, TADalyzer discovers the following information:

Stage Type: TADalyzer tracks the number of active threads at each stage to determine if a stage is bounded or on-demand.

Resource Consumption: Using extended Berkeley Packet Filter (eBPF) [85] and the BPF Compiler Collection [1] tool, TADalyzer attaches hooks on `vfs_read`, `vfs_write`, `socket_read`, `socket_write` and other relevant ker-

nel functions to monitor the I/O and network resources consumed at each stage. The CPU resource consumption is tracked through */proc/stat*. TADalyzer also supports user-defined resources such as database locks or threads in a particular thread pool. However, for custom resources the user is required to annotate the code to tell TADalyzer when the resource is claimed and released.

Intra-Node Data Flow: TADalyzer automatically instruments standard classes that are commonly used to pass requests, such as `util.AbstractQueue` in Java and `std<queue>` in C++, to build data flow between stages within the same node. If a system uses non-traditional data structures to pass requests between stages, the user has to manually annotate the relevant operations (e.g., `put` and `get`) for TADalyzer to correctly track them and build the data flow. Out of the four systems we studied, only HBase uses data structures that we do not automatically monitor – it uses Ring-Buffer [110] to enable lock-free accesses on the WAL entry queue.

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

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

Based on the above information, TADalyzer then generates the TAD. It also gives warning if the NULL stage is responsible for too much resource consumptions; this usually indicates that some resource intensive stages (thus important for scheduling) are not annotated by the user.

3.2.2 Limitations

We now discuss the limitations of TADalyzer. First, TADalyzer relies on run-time instrumentation to collect information, and may miss some in-

formation if the workload supplied to it is not comprehensive enough.

Second, TADalyzer uses kernel instrumentation to track network and I/O activities of each thread, so it only works on systems where an application thread directly corresponds a kernel thread. Most systems based on Java, C++ or other traditional programming languages work well with TADalyzer. However, Riak, which makes use of the Erlang user-level processes [74], requires additional instrumentation to obtain its TAD.

Third, the automatic queue operation instrumentation may miss some data flows. For example, the HBase RPC Read threads use a fast-path mechanism, which directly dispatches an RPC call to an RPC Handle thread without going through the queue to improve data locality. Manual instrumentation is required to capture requests passed in this way.

Finally, contention on application specific resources, such as locks, may greatly impact system performance, but TADalyzer relies on the user to identify such resources.

In summary, the TADs generated by TADalyzer is *correct*, but may be *incomplete*. We use TADalyzer to automatically discover the thread architecture of storage systems, and then argument the architecture based on documentation, code inspection and discussion with the developers.

3.3 HBase/HDFS

The TAD of the HBase/HDFS storage stack is shown in Figure 3.2. HBase is widely deployed in many production environments [46], and achieving schedulability remains difficult despite repeated attempts from both industry and academia [13, 16, 80, 117, 128].

HBase is built on top of HDFS and its data model closely resembles BigTable [32], where data are stored in tables, which have rows and columns. HBase tables are divided horizontally by row key range into *regions*; each *RegionServer* is responsible for managing one or more data regions. Re-

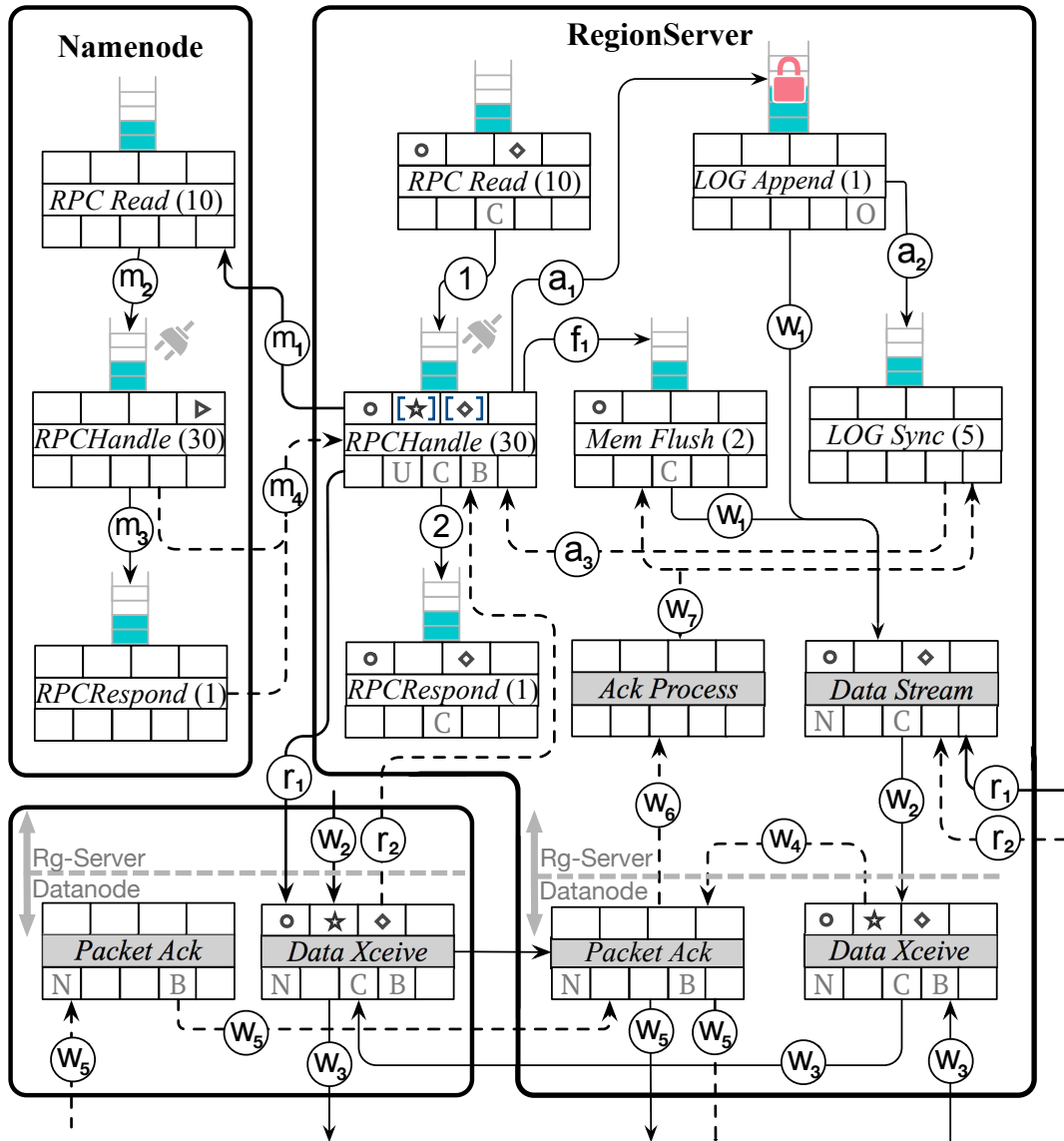


Figure 3.2: HBase/HDFS Thread Architecture. This diagram is based on HBase 2.0.0 and Hadoop 2.7.1.

gionServers usually co-locate with the DataNodes in the HDFS cluster. HMaster assigns regions to different RegionServers, and monitors all RegionServers in the cluster. During reads or writes, the clients first contacts HMaster to find out which RegionServer is serving the concerned data region, and then sends the requests to the particular RegionServer.

3.3.1 Request Flow

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

If the RPC requires metadata lookups, the RPC Handle thread sends a request to the Namenode and blocks until the operation is finished ($m_1 - m_4$); blocking is indicated by the dashed m_4 arrow.

If the RPC wants 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 until the read completes ($r_1 - r_2$). If it is local, a short-circuit mechanism reads data directly within the RPC Handle thread, consuming I/O (HBase still contacts the Datanode to get the file descriptor but the actual reading is local). I/O resource usage in RPC Handle is initially unknown; thus it is marked with a bracket in the TAD.

For operations that modify data, RPC Handle appends WAL entries to a log (a_1) and blocks until the entry is persisted (indicated by the dashed a_3 arrow). The LOG Append thread fetches WAL entries from the queue in the same order they are appended (indicated by lock symbol by the LOG Append queue). LOG Append writes those entries to HDFS by passing data to Data Stream (w_1), which is described in more detail below. LOG Append also sends information about WAL entries to LOG

Sync (a_2), which blocks (w_7) until the write path notifies it of the write completion; it then tells RPC Handle to proceed (a_3).

Along the write path, the Data Stream stage sends data to the Data Xceive stage in an HDFS Datanode, which spawns a thread for each block of data (w_2). The RegionServer and DataNode are usually co-located in one physical node, as indicated by the process boundary lines in the TAD. Depending on the replication level, Data Xceive may pass the data to another datanode, which would spawn another Data Xceive thread to write another copy of the data block (w_3). Data Xceive writes data to disk; for each written data packet, it sends an ack to Packet Ack (w_4). Packet Ack collects acks from downstream datanodes and send acks to either the upstream Packet Ack stage (w_5) or to the issuing client (w_6). Back within the RegionServer, each Data Stream thread also spawns a corresponding Ack Process thread, which is responsible for receiving and processing acks from the Datanode; once the Ack Process thread receives all relevant acks, it notifies the LOG Sync thread that the write is persisted (w_7).

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 ($w_1 - w_7$).

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.

The stages in the Namenode (i.e., RPC Read, RPC Handle, and RPC Respond) are similar to those in the RegionServer. However, RPC Handle in the namenode does not need to invoke operations in other stages; it simply grabs a lock and performs local operations. The TAD shows only the lock resource because it incurs the most contention.

3.3.2 Summary

The HBase TAD shows more than ten complex stages exhibiting different local behaviors (e.g., bounded vs. on-demand), resource usage patterns (e.g., unknown I/O demands), and interconnections (e.g., blocking and competing for the same resources across stages). Understanding such a TAD enables one to identify problematic scheduling in HBase, which we discuss in Chapter 4.

3.4 MongoDB

Figure 3.3 shows the TAD of MongoDB [34], another NoSQL database that is widely used but exhibit very different thread behaviors compared to HBase. MongoDB mostly resembles the traditional thread-based architecture, but its limit on active worker thread numbers and the replication design are strongly influenced by SEDA.

MongoDB stores all data in *documents*, which are JSON-style data structure composed of field-value pairs. Documents of similar structures form a collection, which is equivalent to a table in HBase. MongoDB also stores indexes of the data to support efficient queries. Data in one collection are divided into chunks based on key values among different shards. Each shard is an independent replication set, and consists of one primary and two secondary data nodes. The primary node receives all write operations and records all changes to its operation log (oplog). The secondaries replicate the primary's oplog and apply the operations to their data. Both primary and secondaries can serve read requests.

3.4.1 Request Flow

In MongoDB, a new Worker thread is launched for each client connection. However, only 128 threads are allowed to make progress at any

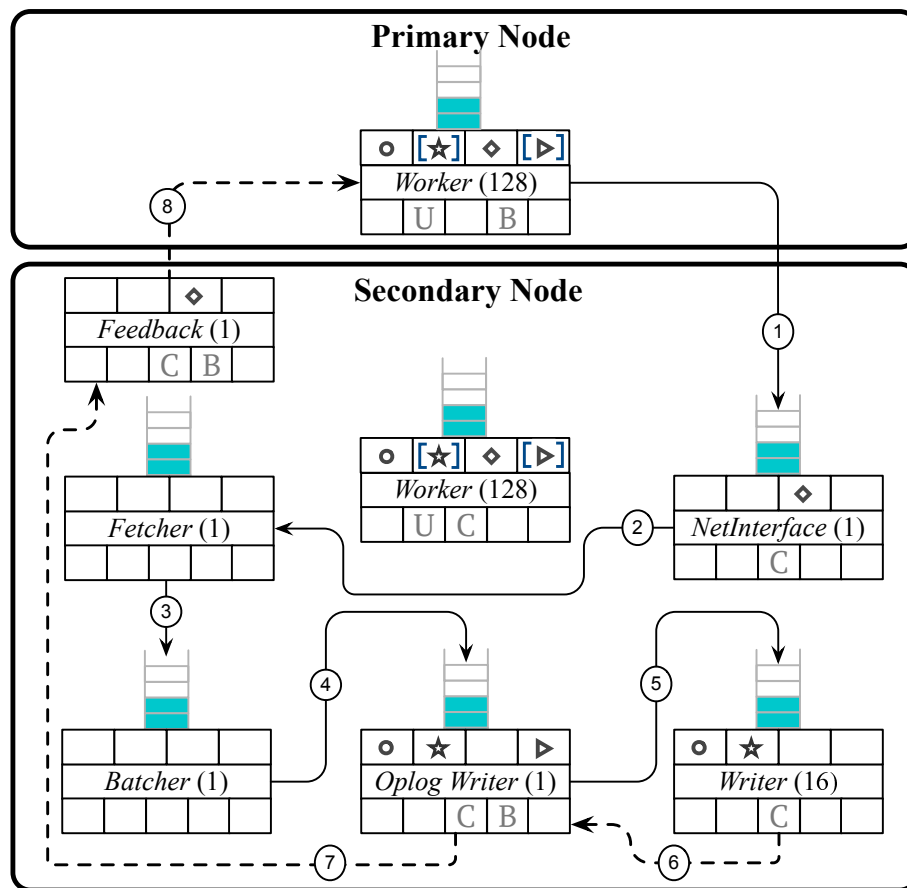


Figure 3.3: MongoDB Thread Architecture. (v3.2.10)

given time, which essentially implements a fixed size thread pool. The Worker threads are responsible for reading requests from the clients, calculating the optimal query plan, carrying out the query plan to retrieve indexes and data, and sending the reply back to the client. During the query processing, the Worker threads may perform I/O synchronously on the underlying storage engine.

When processing write operations, the Worker threads also record the operations in the oplog, which are replicated to the secondary nodes (step 1). Depending on the consistency level, the Worker threads may block

until the writes have been replicated to enough secondary nodes.

The NetInterface thread receives the serialized oplog and passes it to the Fetcher thread (step 2); which parses the oplog and sends the operations to be applied to the Batchter thread (step 3).

The Batchter thread collects the operations and separates them into batches. Operations within each batch have to be applied in order, but different batches can be applied in parallel. These batches are passed to the Oplog Writer thread (step 4).

The Oplog Writer thread first writes operations to the secondary node's own oplog; it then assigns batches to the Writer threads to apply to the main database (step 5). The Oplog Writer thread blocks until the Writer threads finish the batches (step 6), at which point it signals the Feedback thread (step 7), which in turn notifies the primary node (step 8).

At the primary node, if a Worker thread is waiting for the replication to finish, it stops blocking and sends reply to the client.

3.4.2 Summary

In MongoDB, one Worker thread serves a request throughout its lifetime, from reading the request in, to processing it, and to sending the reply back. While conceptually simple, as we will see later (Chapter 4), it causes hard-to-solve scheduling problems.

3.5 Cassandra

The thread architecture of Cassandra is shown in Figure 3.4. Cassandra is a distributed storage system that provides high scalability and fault tolerances on commodity hardwares [73]. In Cassandra, all nodes play identical roles in a "ring" like architecture, and data are replicated to multiple nodes in the ring based on a consistent hashing scheme.

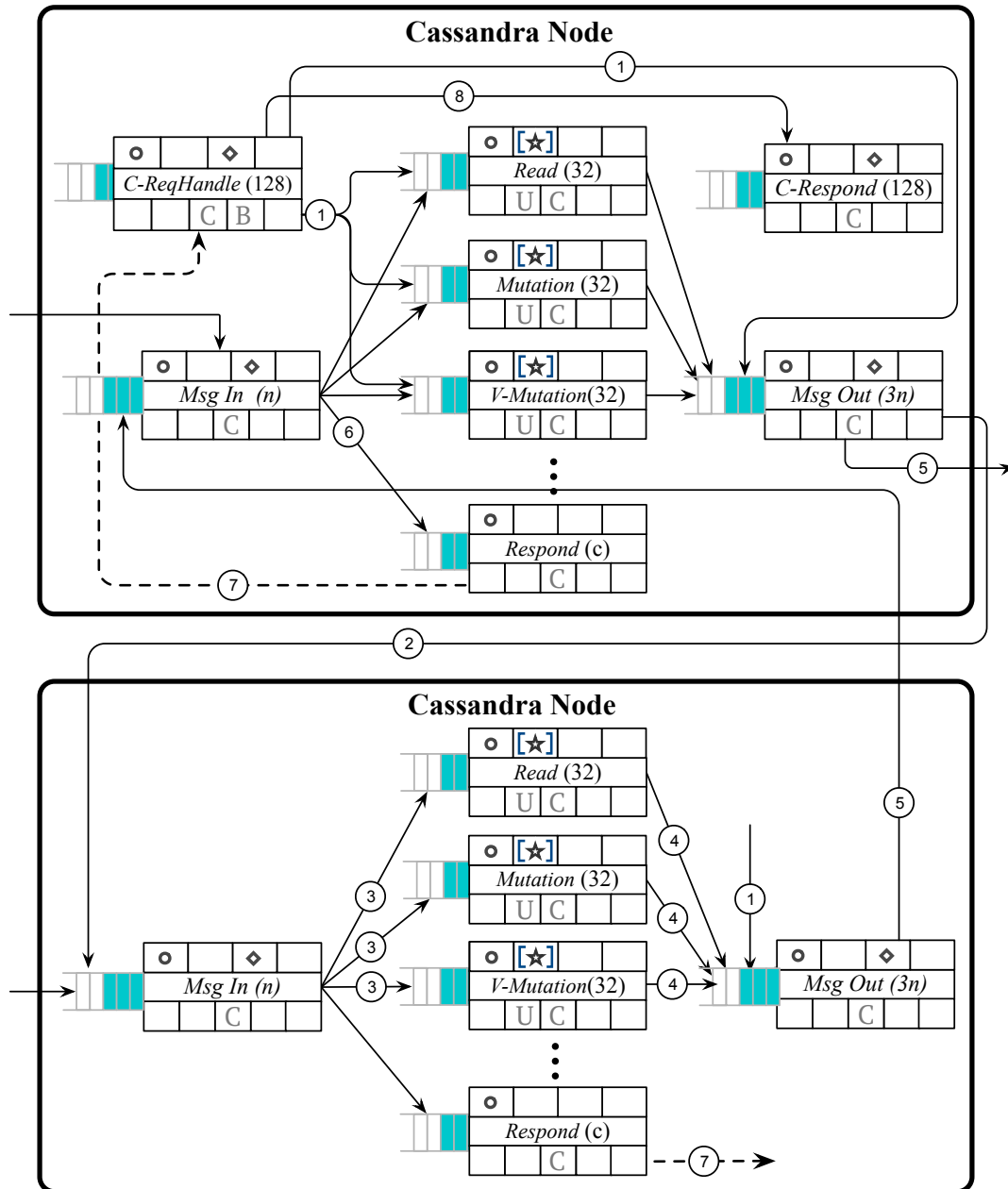


Figure 3.4: **Cassandra Thread Architecture.** (v3.0.10) The ellipsis represent other database processing stages that exhibit similar behaviors and resource usage patterns as the Read, Mutation and View-Mutation stage. c means number of cores; n means number of nodes in a cluster.

3.5.1 Request Flow

When one client sends a query to one of the Cassandra nodes, a thread in the C-ReqHandle stage reads the request in, decodes it, and coordinates its processing; we call this thread the coordination thread of this request.

After parsing a request, the coordination thread first looks up where the relevant data is stored. For local data, the request is directly submitted to the corresponding local processing stages, such as the Read and Mutation stage (step 1). For remote data, the coordination thread passes the request to the Msg Out stage (step 1). It then blocks until the request completes, either on the local or remote node.

The Msg Out stage picks up the requests and sends them through the network (step 2). On the receiving end, the Msg In stage reads the data off the network and de-serializes them. Once finished, the Msg In stage puts the parsed messages in the queue of different processing stage (Read, Mutation, etc.) based on the request type (step 3).

Cassandra has 10 different processing stages: Read, Read-Repair, Mutation, Counter-Mutation, View-Mutation, Gossip, Anti-Entropy, Internal-Response, Tracing, and Misc. We omit most of them in the TAD shown here because they all have similar behaviors. These stages execute the requests, which might include looking up the cache, performing I/O, and compressing/de-compressing the data. After completing a request, the processing stages generate a response and pass it to the Msg Out stage (step 4).

The Msg Out stage in the remote node sends the response back, which is received by the Msg In stage (step 5). The Msg In stage passes the response to the Respond stage, who is responsible for executing any callbacks associated with the request completion (step 6). Finally, the coordination thread in the C-ReqHandle stage is notified and finishes blocking (step 7); it passes the response to the C-Respond stage, who is responsible for serializing the response and sending it to the clients.

3.5.2 Summary

Cassandra closely follows the standard SEDA architecture, where all activities are managed in bounded stages. These stages are divided based on their functionalities; there are dedicated stages to handle reads, mutations, counter-mutations, etc. As we will discuss later (Chapter 4), such a division weakens SEDA's strength on explicit resource management, and causes scheduling problems.

3.6 Riak KV

The TAD of Riak KV is shown in Figure 3.5. Riak KV [71] (hereafter just Riak) is a newly emerged distributed NoSQL database based on the functional programming language Erlang [74]. Similar to Cassandra, Riak uses consistent hashing to divide data into partitions and places multiple partitions in one physical node. Built on top of the Erlang VM (virtual machine), Riak relies heavily on the light-weighted processes (referred as threads in the following description) and transparent IPC mechanisms the Erlang VM provides.

3.6.1 Request Flow

When clients issue requests to one of the Riak nodes, a new Req In-Out thread is created for each new client connection to read from/write to the connection, and to encode/decode the messages. After decoding a request, the Req In-Out thread passes the request to the Req Process thread (step 1), which is also spawned on demand for each new client connection. The Req In-Out thread then blocks until the response of the request is available.

The newly created Req Process thread looks up in which partition the requested data is, and sends the request to one or more of these partitions

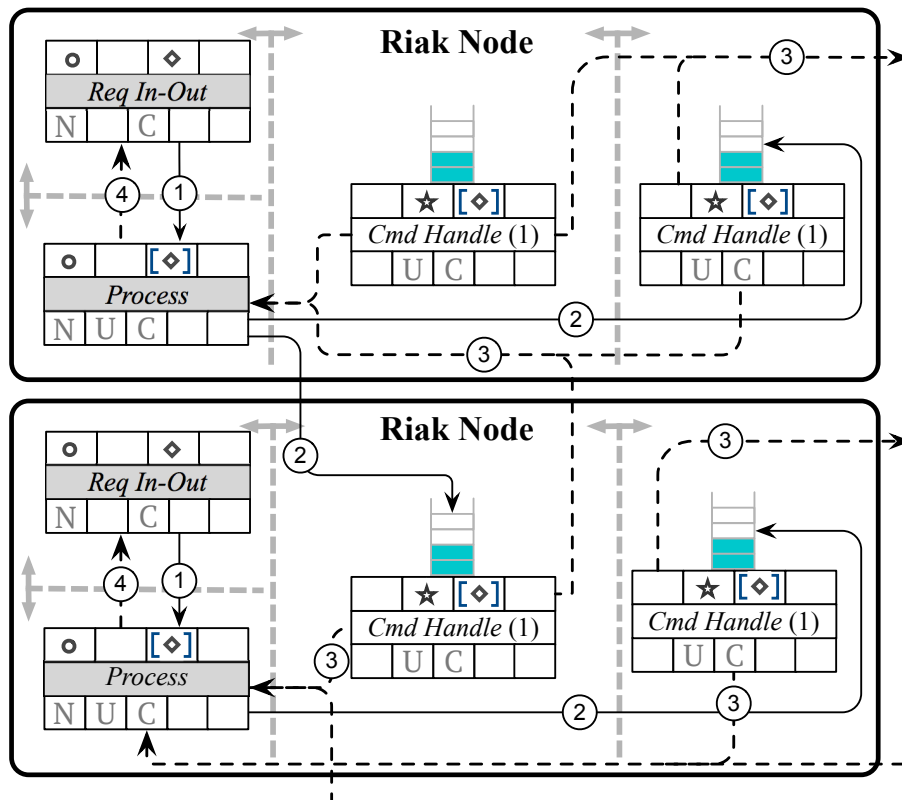


Figure 3.5: Riak Thread Architecture. (v2.1.4)

(step 2). It then blocks until the request is completed by the partitions.

There is one Cmd Handle thread for each partition, which is responsible for performing I/Os. Upon completion of the I/O, the Cmd Handle thread sends the response back to the issuing Req Process thread (step 3).

After hearing back from all the Cmd Handle threads it sends requests to, the Req Process thread generates the response to the client and passes it to the Req In-Out thread (step 4). The Req In-Out thread encodes the response, sends it off the network and waits for another request from the client.

3.6.2 Summary

Riak is another system that closely follows the SEDA design. However, instead of using bounded stages, Riak heavily uses on-demand stages and relies on the underlying Erlang VM to schedule the potentially large number of threads [43]. Riak is also oblivious on whether a partition it accesses is local or remote; it contacts the Cmd Handle thread responsible for that partition using the same IPC mechanism. Though simplifying programming, these designs cause additional challenges when enforcing scheduling on Riak, which we will discuss later (§4.5.3).

3.7 Conclusions

In this chapter we introduce the Thread Architecture Diagrams as a general way to describe the thread behavior and interactions of a system, which in turn determines the system's *schedulability*. We show the TADs of four widely used systems: HBase/HDFS, MongoDB, Cassandra, and Riak, and describe how they work under the lens of TAD. Next we will use the tools we developed in this chapter to study the scheduling problems on distributed storage systems.

4

Maat: Toward Schedulability on Distributed Storage Systems

Request scheduling lies at the heart of scalable storage systems. However, introducing scheduling control into modern storage systems is challenging due to their complex thread architectures. Equipped with the tool we developed in Chapter 3, we now turn to solving such scheduling challenges.

System developers rarely have the luxury of constructing a new service from scratch that can provide a new scheduling policy or goal. Instead, developers often retrofit new scheduling policies into existing architectures containing a number of limitations and flaws. We have two resulting questions. First, which types of thread architectures most naturally enable new scheduling policies to be implemented? Second, when problems do exist in a thread architecture, how can those problems be most easily remedied?

We begin by considering an ideal thread architecture. To realize a global scheduling policy in a distributed storage system, one first needs to translate the global policy into local scheduling plans to be enforced at key points of the system; the local plans are then implemented by local schedulers inserted at these points. For example, to achieve global fairness, one might allocate local client share at each node to match the clients' demand; these local shares are then enforced by weighted fair queuing

schedulers at the individual storage nodes, similar to the approach proposed in Pisces [103].

Since Maat is the ancient Egyptian concept for order, balance, and ideal [67], we say a system adheres to the Maat principle if it satisfies the following three conditions that make realizing a scheduling policy easy: **Completeness** – the system provides necessary scheduling points so that a global policy *can* be translated into local scheduling plans at these points. **Local enforceability** – the local scheduling plans *can* be implemented. At each scheduling point, the system provides both enough *information* and *control* to the local scheduler to make implementing the plan possible. **Independent scheduling** – the decisions made by one local scheduler do not have unexpected effects at other scheduling points.

Unfortunately, most systems violate the above conditions in one or more ways, causing scheduling difficulties. In particular, we identify five common problems exhibited in modern distributed storage systems that cause violations of the ideal Maat scheduling conditions: *no scheduling points*, *unknown resource usage*, *hidden competition*, *blocking*, and *ordering constraint*. We illustrate how each of the problems causes Maat violation and leads to scheduling difficulties.

We also show how to fix an existing stage so the system is Maat-compliant. In general, a stage can adhere to the Maat principle in one of two ways: *directly* with a thread architecture that naturally avoids Maat violation or *indirectly* by adding control logic that adjusts the stage behavior based on additional informations (e.g., knowledge about past requests in this stage or other stages and their resource usage) to mitigate the problem. There are different tradeoffs for direct and indirect solutions. We apply the Maat principle to the most complex system that we studied, HBase/HDFS; through a combination of direction and indirection methods, we show that HBase-Maat can be transformed to provide schedulability.

The rest of this chapter is structured as follows. First we introduce

the TAD-based simulation framework that allows one to emulate a system's behavior with high fidelity to study its scheduling properties (§4.1). We then discuss the Maat principle, centered around the five fundamental scheduling problems (§4.2); we use simulation to demonstrate these problems. Next we apply Maat to HBase/HDFS, showing via simulation (§4.3) and implementation (§4.4) how to make said system schedulable; we then briefly discuss schedulability of other systems and show their scheduling problems via simulation (§4.5). Finally, we conclude (§4.6).

4.1 TAD Simulation

Before diving into the Maat scheduling principle, we first build a simulation framework to facilitate the understanding of systems' scheduling behavior. Based on `simpy` (a Python package for event-driven simulation) [84], the simulation framework provides basic building blocks such as requests, threads, stages, resources and schedulers. Using the system TADs as blueprints, one can assemble the stages in different ways to form various thread architectures and insert different schedulers at each scheduling point. The framework then simulates how requests flow through the stages and consume resources, and report detailed statistics (e.g., client throughput, latency, and resource utilization).

In this section, we describe the design and implementation of this simulation framework, and its limitations. We show the fidelity of the simulation later (§4.4), where we compare our simulation results to the results obtained on real implementations.

4.1.1 Design

The architecture of the TAD simulation framework is shown in Figure 4.1; it is based on three key abstractions: Request, Resource, and Stage.

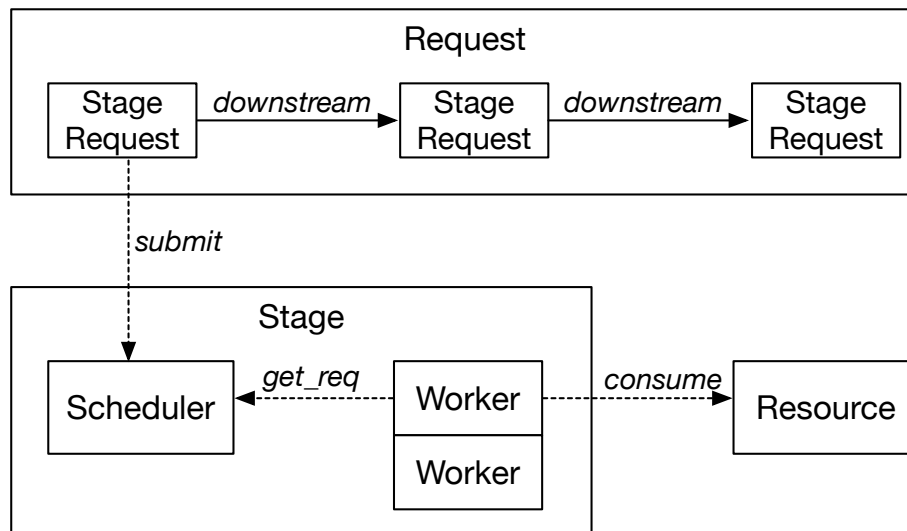


Figure 4.1: **Simulation Framework Architecture.**

A Request encodes how this request flows through different stages in the system; the processing required at each stage is represented by a StageRequest. Each StageRequest has a ResourceProfile that describes the amount of resources to be consumed during its processing, and also points to a list of downstream StageRequest's to be submitted next. This downstream list indicates the immediate next stages the Request flows through. After the processing of a StageRequest, all StageRequest's in its downstream list are submitted. A StageRequest may block on certain events (e.g., the *done* event) on the downstream StageRequest's before it can finish (raising its own *done* event). A Request is considered finished when all specified StageRequest's finish; note that it needs not to be all the StageRequest's included in this Request, because some processing can be done after the acknowledgement to the client.

Resource can be used to model any kind of resources, e.g., cpu, I/O, network. One can specify the rate and parallelism of a resource. For example, a 1GHz cpu with 8 cores has a rate of 1 and a parallelism of 8. Resource provides an interface to consume a certain amount of resource;

the caller blocks until the resource consumption is done.

A Stage is a set of workers (threads) that continuously pick StageRequest's submitted to it and process them. During the processing, the worker consumes resources based on the StageRequest's ResourceProfile. The worker then submits the downstream StageRequest's to their designated stages, and blocks on events on the downstream StageRequest's if necessary. A Scheduler encodes the logic a Stage uses to pick up requests to process from all the requests that are submitted to it. Different scheduling policies can be implemented as different schedulers, which are then plugged into Stages to control the Stages' behavior. The framework provides commonly used schedulers such as FIFO, DRF and priority schedulers; the user can also define other schedulers based on his/her needs. Scheduling is done based on the cost of the StageRequest; depending on its scheduling goal, a Scheduler might calculate the cost differently, thus each Stage also supplies its Scheduler a CostFunction, which is used to map a StageRequest to its cost.

Request	setFirstStageReq(stageReq)
StageRequest	submit(stage) addResourceConsumption(resource, amount) addDownstreamRequest(stageReq) addBlockingEvent(event)
Resource	consume(amount)
Stage	setScheduler(scheduler) setCostFunction(costFunction)
Scheduler	getReq()

Table 4.1: **Simulation APIs.**

Table 4.1 summarizes important APIs provided by the simulation framework. Using these APIs, one can construct a system as simple or complex as one wants. In particular, the framework is capable of simulating any behavior described in a TAD. In the rest of this chapter we will show how

targeted simulation based on the TAD of a system can provide insights in its scheduling problems.

4.1.2 Simplifications

When designing the simulation framework, we made some conscious choices to keep it simple and easy to use, even when sometimes the simplicity comes at the cost of deviation from the real world. We now briefly discuss the simplifications.

First, we do not consider the cost of creating a thread and switching between threads. Such cost varies significantly depending on the thread implementation and the workloads. Moreover, the scheduling problems we discuss are present even when these activities are completely free.

Second, resources in our simulation have fixed rate and parallelism; we do not capture complex resource performance properties such as how disk throughput changes on the randomness of the requests. Moreover, all resources we model are completely stateless; we do not consider resources such as cache whose usage depends on its current state. Though we do not treat cache as a resource and study how to share cache among different clients, we do take into account the effect caching has on other resource usages (e.g., a request that hits cache does not consume I/O resources).

Finally, in our simulation resource consumptions are non-preemptive. Once a thread in a Stage starts accessing a resource on behalf of a `StageRequest`, it is not interrupted until it finishes consuming the amount specified in the `ResourceProfile` of the `StageRequest`. Such preemptions, even when they happen, are rarely in the control of the application scheduling logic. For example, CPU preemption is controlled by the operating system.

4.2 The Maat Principle

In this section we present the Maat principle in detail. Our analysis centers around five Maat-violation problems that we posit are at the core of inadequate scheduling: a lack of local scheduling points, unknown resource usage, hidden competition between threads, uncontrolled thread blocking, and ordering constraints upon requests.

To illustrate the Maat principle more clearly, we begin by focusing on systems with only a single problem; in Section 4.3 we consider the HBase TAD in which multiple problematic stages are interconnected. We use simulation to show both the symptoms of the scheduling problems and their root causes. Unless otherwise noted, all simulations in this section use a common configuration, which is summarized in Table 4.2.

Goal	Isolate C1 from the behavior change of C2.
Clients	Two closed-loop clients, C1 and C2, continuously issue requests. C1 has 40 parallel threads; thread count of C2 varies.
Resources	Stages in 1 node with 1 GHz CPU, one disk with 100 MB/s bandwidth, and 1 Gbps network.
Schedulers	Weighted fair queueing for dominated resource fairness (DRF) [47]; equal weight across clients.

Table 4.2: **Simulation Configuration.** *Common configuration across the simulations in § 4.2.1-§ 4.2.5.*

4.2.1 No Scheduling Points

The completeness condition dictates that each resource-intensive stage in a thread architecture provides local scheduling. An on-demand stage with significant resource usage suffers from the *no scheduling points* problem because the system has no scheduling point to control resource accesses within this stage; these unregulated resource-intensive activities can thus violate the system’s overall scheduling goal (e.g., fairness). One

can identify the no scheduling points problems in a TAD by looking for stages with shading and resource usage (e.g., in HBase, Data Stream and Data Xceive suffer from no scheduling points).

Figure 4.2(a) shows a simple TAD with two stages, the second of which has no scheduling points (an on-demand stage with intensive I/O). The scheduler for the first stage (Q1) attempts to enforce DRF [47], but is unsuccessful: as client C2 issues requests with more threads, C2 receives more I/O resources (up to 90%) and the throughput of C1 declines. The problem occurs because Q1 scheduling is irrelevant given that requests are not bottlenecked at this stage. The Req Handle threads quickly finish processing the request, pass them to the I/O stage and return to ask Q1 for more requests. As shown in Figure 4.2(a), the average length of Q1 is zero and it therefore cannot reorder requests. Meanwhile, there are many requests contending for I/O in the second stage, which the first scheduler has no control over and no scheduling points is provided at.

Figure 4.2(b) shows an indirect solution in which the thread architecture remains the same, but control logic is added to limit the number of requests sent from the Req Handle stage to the I/O stage. To achieve this, Q1 enforces rate limiting instead of DRF. Specifically, information from the I/O stage is collected to estimate per-client I/O demand, utilization, and additional capacity using an Additional-Increase-Multiplicative-Decrease (AIMD) algorithm [33]; based on the estimate demands and capacity, each client's weight is translated to a rate limit, pushed to Q1 to enforce fairness. System (b) is able to enforce global scheduling despite the lack of a scheduling point at the I/O stage because Q1 gives up work conservation: Q1 postpones requests from C2 even when there are idle Req Handle threads. Because of this, any work-conserving scheduling policy such as fair queuing or priority-based scheduling would have to be emulated at the control plane. Many indirect approaches that share information are possible; for example, Retro [80] similarly translates different scheduling

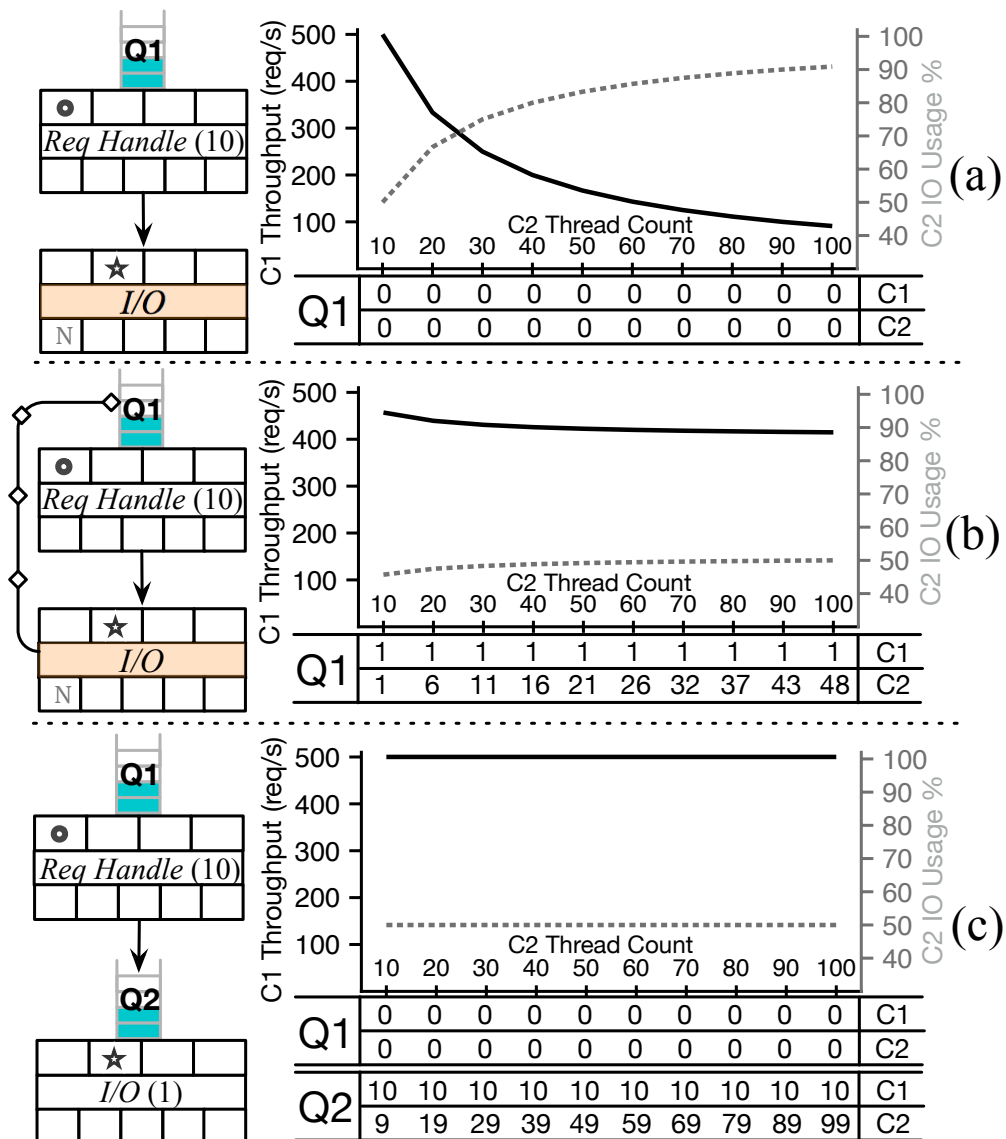


Figure 4.2: **The No Scheduling Problem.** Each client request requires 100 μ s CPU time and 100 KB I/O, making I/O the bottleneck. The average number of requests per client waiting to be scheduled at each scheduling point is shown below.

policies into rate limits.

Figure 4.2(c) shows how to solve the problem directly by adding 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. The direct approach simply and naturally ensures fairness and isolation of the two clients with no need for information sharing across stages.

4.2.2 Unknown Resource Usage

A stage has *unknown resource usage* if 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 usage patterns and the scheduler does not know this ahead of time. Unknown resource usage forces the scheduler to make decisions early, before important resource information is available, thus violating the local enforcement condition (not enough information is provided to the scheduler to make scheduling decisions). Unknown resource usage is denoted in a TAD by stages with square brackets around resources. In HBase, the RPC Handle stage exhibits unknown resource usage on I/O due to the short-circuited reads it might perform.

Figure 4.3(a) shows a single stage with unknown I/O resource usage (note the bracket on the I/O resource). Even though Q1 allocates the I/O resource, it does not know whether a request requires I/O at the time of scheduling because a request may hit cache. When C2 issues a mix of cold- and hot-cache requests, Q1 schedules C2-cold and C2-cached in the same way, causing low CPU utilization and low throughput for C2-Cached.

System (b) solves this problem with the indirect method of specula-

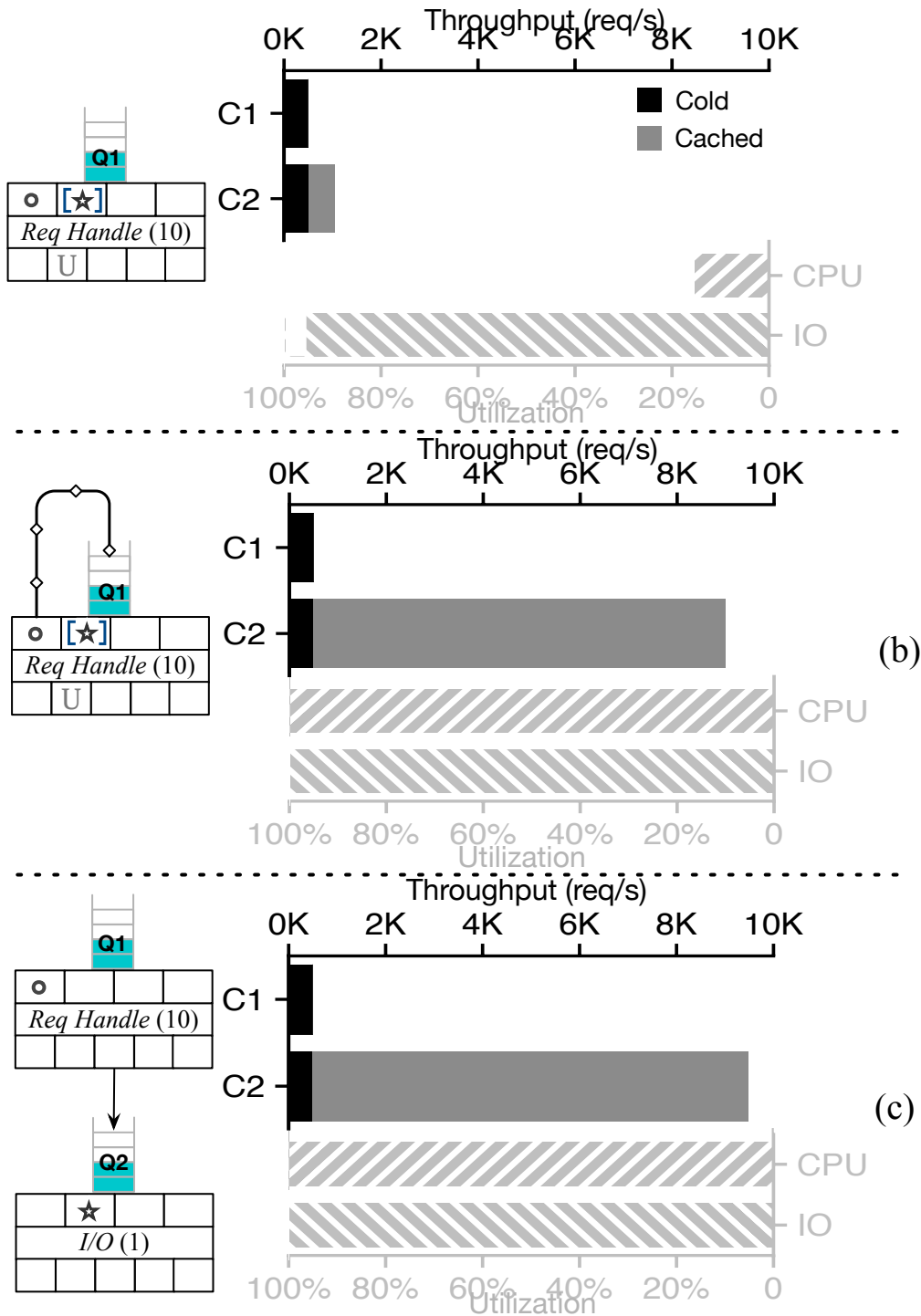


Figure 4.3: The Unknown Resource Usage Problem. Both C1 and C2 issue requests that require 100 us CPU time and 100 KB I/O. C2 also issues cached requests that requires only 100 us time but no I/O. In (a) and (b) the Req Handle threads first look up the cache when serving a request, and perform I/O if it is a cache miss. In (c) a separate I/O stage performs I/O if the Req Handle stage cannot serve a request from cache.

tive execution. While many approaches are feasible, the simulated approach speculatively executes a waiting request when the CPU is idle. If during the speculative execution, the request is found to require I/O, the request is aborted and put back on the queue where it is subjected to normal scheduling. For the single-stage system (b), speculative execution provides high throughput for C2-cold, but must abort some requests. Speculative execution works best for predictable workloads and relatively simple resource usage patterns; if a stage may potentially perform I/O, initiate network connections, and contend for locks, predicting whether a request will use each resource becomes much harder and less accurate.

System (c) solves the unknown resource problem directly by splitting one stage 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 has exact knowledge about the size and location of the I/O, enabling Q2 to make better scheduling decisions.

4.2.3 Hidden Contention

When multiple stages with independent schedulers compete for the same resource, they suffer from *hidden contention* which impacts overall resource allocation in unexpected ways, causing violation of the independent scheduling condition of the Maat principle. In a TAD, one can identify hidden contention by stages within a node boundary that have separate queues but the same resources in the resource usage boxes. Hidden contention is ubiquitous in every system we investigate, because some contention is difficult to avoid (e.g., most stages use CPU). In HBase, both RPC Handle and Data Xceive might compete for the same I/O resource; multiple stages compete for CPU and network.

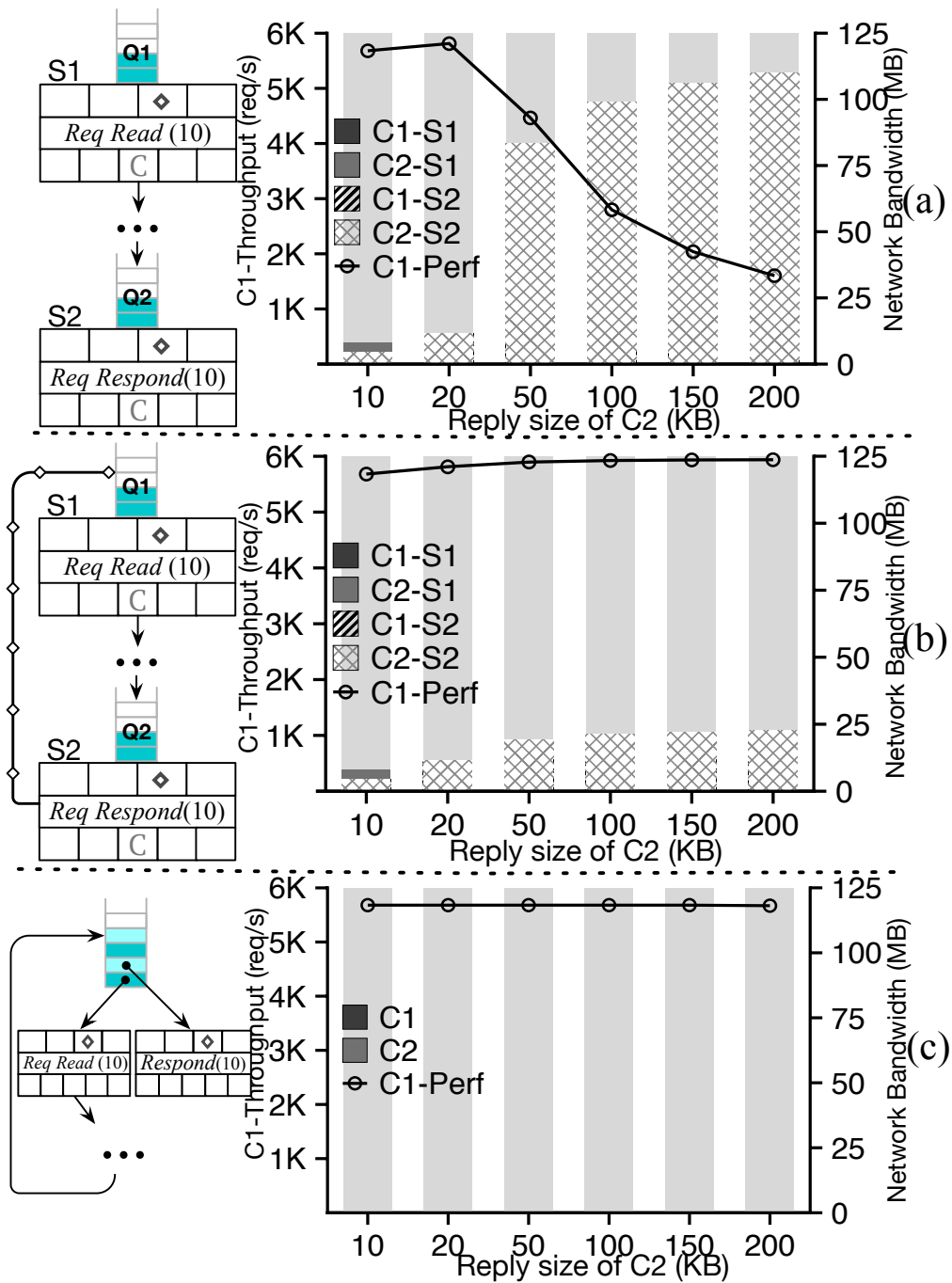


Figure 4.4: **The Hidden Contention Problem.** *C1* sends 1 KB requests and receives 10 KB replies; *C2* also sends 1 KB requests but its reply size varies (shown in the x-axis). The left y-axis shows throughput of *C1*. The right y-axis shows the total network bandwidth and the bandwidth each stage 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 4.4 shows a two-stage system with the network as the source of hidden contention; one stage reads requests and the other sends replies. Enforcing fairness at each stage does not guarantee fair sharing at the node level. Note that both clients have much larger reply size than request size, so S2 (the Req Response stage) consumes more than 90% of the total network bandwidth and S1 (the Req Read stage) consumes less than 10%. Yet scheduling at S2 does not ensure 90% of the network resource is fairly shared across C1 and C2. 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. One can see from Figure 4.4-a that with larger C2 reply sizes, the scheduler S2 is frequently forced to schedule C2 because there are no requests from C1. C2 in the second stage effectively monopolizes the network and prevents S1 from using the network; this causes fewer requests to be completed at S1 and flow to S2, further limiting choices available to S2. Hidden network contention between the two stages leads to this cycle and causes unfair scheduling.

System (b) solves the problem indirectly by sharing information about network usage across the two stages. In the simulated approach, if a stage uses excessive resources, it backs off by reducing its thread count, allowing the other stage to catch up. The figure shows that system (b) allows S1 to process more requests, thus giving S2 more freedom when scheduling, which leads to effective isolation of C2. Although S2 is sometimes forced to schedule C2, S2 can compensate by favoring C1 later.

System (c) solves the problem directly by having the two stages share the same scheduling point. Both the Req Read and Req Respond stage use Q1 for requests, though each handles different request types. This change gives Q1 full control of the network and enables it to isolate C1 and C2 perfectly.

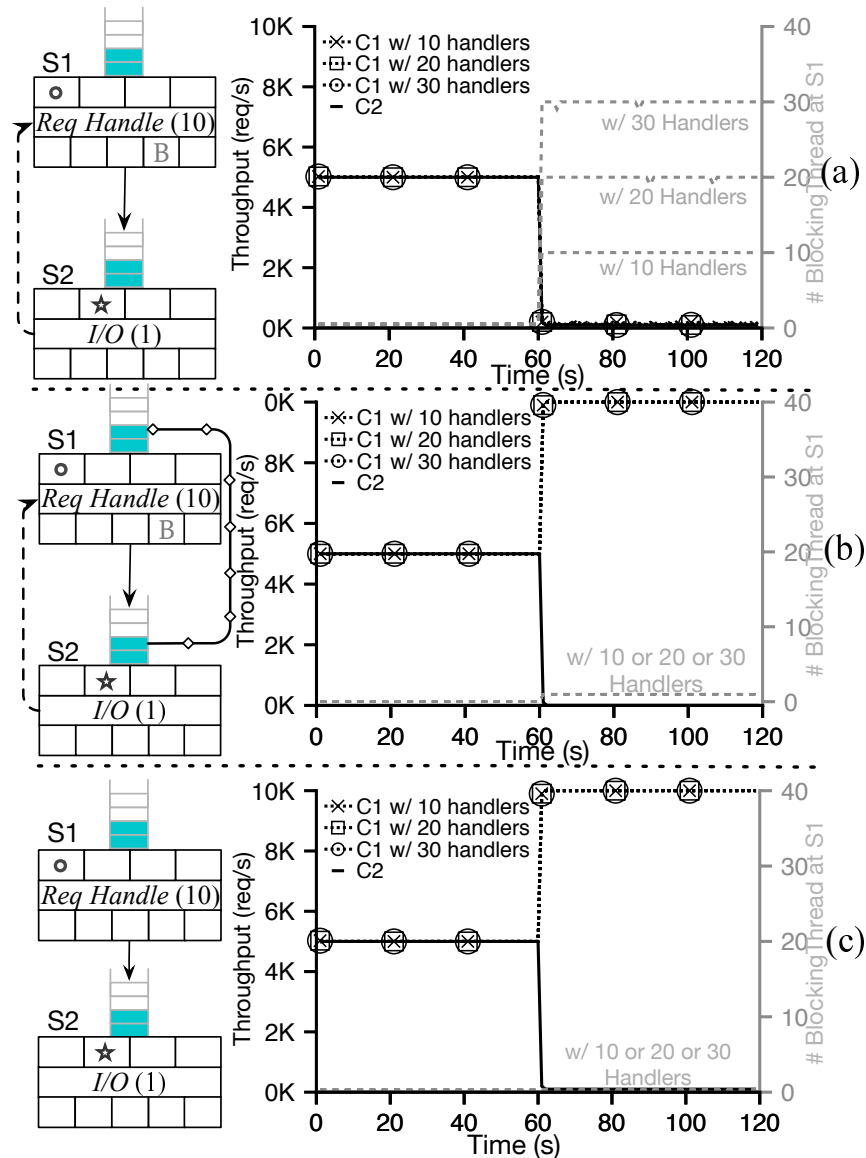


Figure 4.5: The Blocking Problem. Initially both C1 and C2 issue requests that require 100 μ s CPU time and can be completed within the Req Handle Stage. At time 60, C2 switches to an I/O intensive workload where each request requires 100 μ s CPU time at the Req Handle stage and 100 KB I/O at the I/O stage. C1 continues to issue CPU-only requests. The simulation was run with 10, 20, and 30 threads at the Req Handle stage, shown in "w/ 10" etc.

4.2.4 Blocking

A system has a *blocking* problem if a bounded stage may block on a downstream stage and requests in the bounded stage may follow different paths. In an ideal system, even when some requests are blocked, each stage allows other requests to make progress; a problem occurs if there are no unblocked threads to process other requests. Blocking is a problem because it violates the independent scheduling condition of the Maat principle and forces upstream stages to account for downstream progress. In a TAD, blocking is indicated by bounded stages with dashed arrows pointing to them (this does not necessarily lead to the blocking problem, which requires multiple data paths, but is a strong indicator). In HBase, RPC Handle blocks on namenode lookups, HDFS reads, and LOG appends.

Figure 4.5(a) shows a system with the blocking problem at the Req Handle stage. Requests entering Req Handle have two paths: they may complete in this stage or cause a thread to block on the I/O stage. Initially both C1 and C2 receive high throughput as they issue CPU-intensive requests without blocking; however, when C2 switches to an I/O-intensive workload, the throughput of both C2 and C1 drop to the disk rate. The Figure shows that all threads in Req Handle are blocked on I/O, leaving no threads to process C1 requests that do not require I/O. Increasing the threads at the Req Handle stage from 10 to 30 does not help: it only leads to more blocked threads.

Figure 4.5(b) shows an indirect solution where the system tries to avoid blocking with a feedback loop: information about the congestion level and estimated queuing delay for each client at the downstream stage is passed upstream. S1 avoids scheduling a request if it anticipates excessive blocking, reserving threads for more useful work. Using this information-based anticipation, system (b) keeps the number of blocked threads low and provides high throughput to C1.

In system (c) blocking is directly eliminated by making the Req Han-

dle stage asynchronous. No threads block and all perform useful work, leading to high throughput for C1.

4.2.5 Ordering Constraint

When a system requires the requests at a stage to be served in a specific order to ensure correctness, it has the *ordering constraint* problem. For example, many storage systems use Write-Ahead Logging (WAL), which requires the writes to the log to occur in sequence. Ordering constraint violates the local enforcement condition of the Maat principle because the local scheduler cannot re-order requests as desired, leaving the scheduling framework with fewer or no choices. One can identify ordering constraints in TAD by looking for the lock symbols on the scheduling points. HBase imposes an ordering constraint at the Log Append stage.

Figure 4.6(a) shows a two-stage system with ordering constraints on the second stage. The scheduler enforces 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.

The ordering problem can be mitigated with indirect methods that share information across stages. In system (b), the two stages coordinate to ensure that there are never more than 10 requests in the LOG Append queue. Although requests in the LOG Append stage must still be served in order, the number of possible low-priority requests is now bounded. The Figure shows that the latency of C1 increases initially, but is bounded eventually.

System (c) directly solves the problem by separating requests from different clients into different streams; even though requests within a stream are still serviced in order, the scheduler can choose which stream to serve

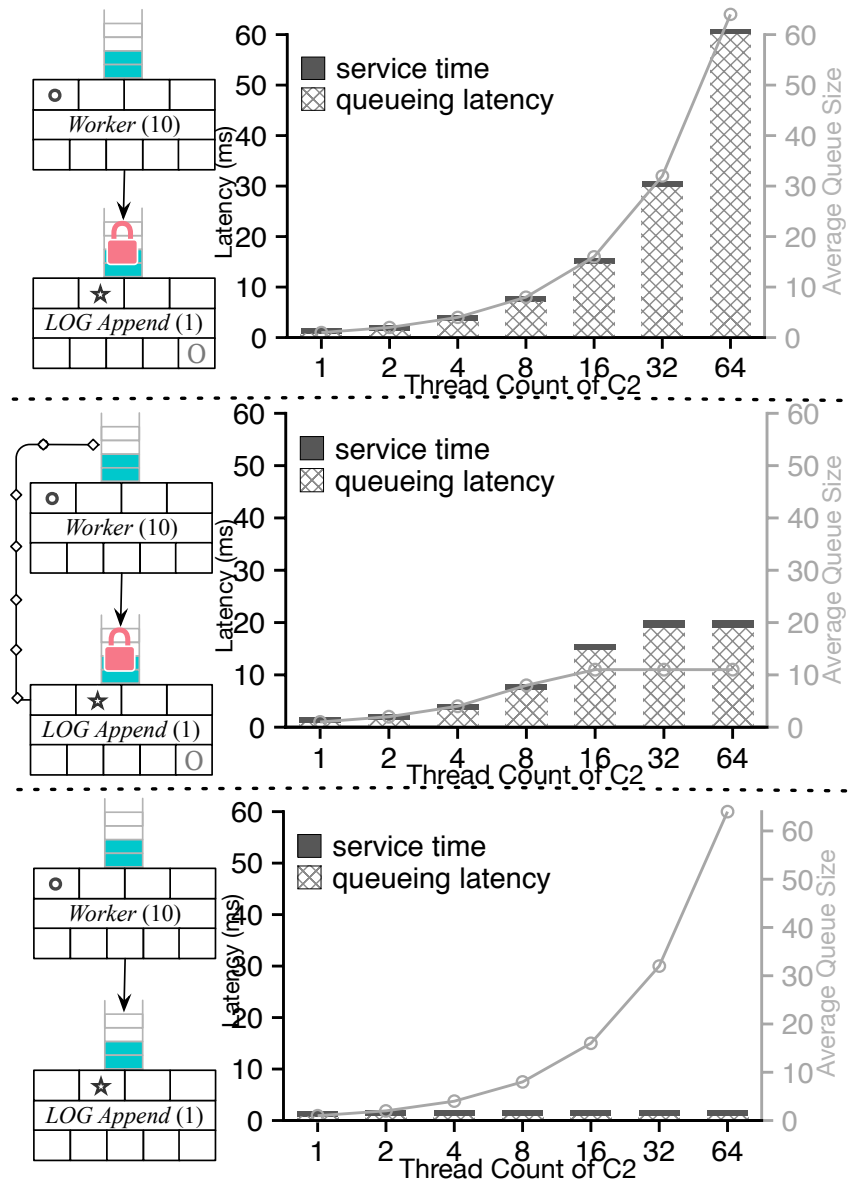


Figure 4.6: **The Ordering Constraint Problem.** High priority C1 issues requests in burst; low priority C2 steady issue requests with more threads (shown in the x-axis). Each request requires 100 us CPU time at the Worker stage, and 100 KB I/O at the I/O stage. The left y-axis shows the average latency of C1's request; the right y axis shows the average queue size of the LOG Append stage.

Problem Identifiable on TAD	Maat Condition Violated
No Scheduling Points	Completeness
Unknown Resource Usage	Local Enforceability
Hidden Contention	Independent Scheduling
Blocking	Independent Scheduling
Ordering Constraint	Local Enforceability

Table 4.3: **Summary of Scheduling Problems Identifiable on TAD and Their Maat Violations.**

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.

4.2.6 Summary and Discussion

In this section we identify five categories of scheduling problems that violate the Maat principle and that can be identified on a TAD. Table 4.3 summarizes how each problem violates the ideal scheduling conditions specified by Maat.

These scheduling problems can be fixed with indirect and direct methods. Indirect methods add information within and across stages to help a thread architecture work around inherent structural flaws of its stages. The advantage of indirect approaches is that they usually involve minimal changes to the thread architecture, which implies low engineering effort. However, adding information flow and complex feedback logic can add overhead and increase convergence time. Furthermore, indirect solutions often rely on stable workload characteristics, so that resource demands and path usage is predictable. The direct method has the opposite strengths and weaknesses. Choosing an indirect or direct approach for a stage will be specific to each system and the complexities of its code base, thread architecture, scheduling goals, and workload.

Previous researchers have proposed frameworks that ensure fairness [103], enforce SLOs [117], provides reservations [128], and more [80, 102, 106]. Many frameworks focus on calculating a scheduling plan to achieve a global policy. For example, Pisces [103] discusses how to allocate local weight at each scheduling point to achieve global fairness; Cake [117] proposes a feedback loop to adjust local scheduler behavior to provide latency guarantees; Argus [128] uses a centralized controller to calculate resource reservations for each individual scheduling point. However, a problematic thread architecture may prevent these plans from being effective.

To achieve scheduling goals in real storage systems, proposed frameworks must overcome the five categories of problems we have identified. Table 4.4 summarizes how we believe some previous frameworks *could* solve each problem (although the solutions may not be discussed explicitly in their documentations). We believe that previous systems have innovative scheduling plans for various scheduling goals, and when realizing these plans, they may coincidentally provide solutions for some scheduling problems. However, none have addressed all five problems in a systematical way. For example, Libra [102] provides fair I/O allocation by delaying threads that call file system APIs; while this mechanism provides a single scheduling point across multiple stages and could remove hidden contention, it does not solve other problems. Retro [80] applies rate limits to multiple scheduling points to emulate different scheduling policies, but does not provide enough information or control to the local schedulers when presented with unknown resource usages and ordering constraints.

Methods	Problems				
	N	U	C	B	O
Cake (2012) [80]	D (partly)	✘	✘	I	✘
Pisces (2012) [103]	✘	✘	✘	✘	✘
Libra (2014) [102]	D (partly)	✘	D	✘	✘
Argus (2015) [128]	✘	✘	✘	✘	I
Retro (2015) [80]	I&D	✘	I	I	✘
Maat	I&D	I&D	I&D	I&D	I&D
HBase-Maat	D	I	I&D	I	I

Table 4.4: **How Different Scheduling Framework Meet Scheduling Challenges.** *D: Direct solution, I: Indirect solution, ✘: no solution proposed.*

4.3 Applying Maat to HBase

The five problems we describe can potentially cause ineffective scheduling, but how these problems manifest themselves depends on interactions across multiple stages, available resources, workload, and the desired scheduling policy. We now apply the Maat principles and investigate the tradeoffs involved for a real-world system, HBase/HDFS. The TAD of HBase is shown in Figure 3.2.

Complex systems such as HBase contain multiple problems and the presence of one problem may hide others. For example, an on-demand stage does not encounter the blocking problem because it spawns a new thread to handle more requests; however, if a scheduling point is added to this stage to fix the no scheduling problem, the blocking problem may manifest. Therefore, we solve the largest problems first and then use our solution to identify new problems that arise. For terminology, we start with Maat-[?][?][?][?], which corresponds to the original HBase. After we choose a direct or indirect solution, we replace ? with d or i.

We simulate a wide range of solutions for the HBase TAD, and then implement our chosen solutions to both validate our TAD model and to discuss the actual engineering effort. Our simulated HBase cluster has 8

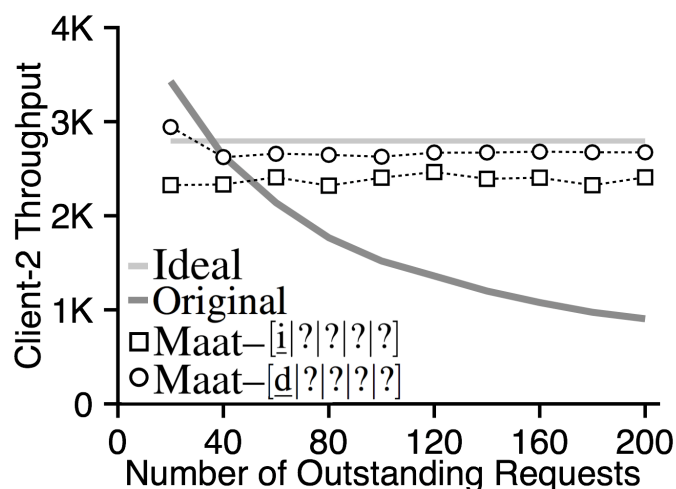


Figure 4.7: **No Scheduling Solutions.** Clients keep issuing (uncached) Gets, each of which incurs 128 K I/O at the Data Xceive stage. C1 has 40 threads issuing requests in parallel; number of threads of C2 increase from 40 to 200.

nodes; one master node hosts the HMaster and Namenode, and 7 slave nodes host RegionServers and DataNodes. Each node has one 1 GHz CPU and one disk with 100 MB bandwidth, and is connected via 1 Gbps network. Client identifiers are propagated across stages with requests, so each scheduler can map requests back to the originating client.

4.3.1 No Scheduling

Problem: The Data Xceive and Data Stream stages consume significant resources but export no scheduling points; RPC Read and RPC Respond have only hard-coded scheduling logic.

We investigate both direct and indirect solutions. For the direct solution (Maat-[d|?|?|?|?]), we add scheduling points to the Data Xceive and Data Stream stages.

Indirectly solving the no-scheduling problem with rate limiting (Maat-[i|?|?|?|?]) is much more complicated. RPC Handle, Data Stream, and Data

Xceive stages on different nodes can all issue requests to Data Xceive on one DataNode; similarly, both LOG Append and Mem Flush may issue requests to the Data Stream stage. Thus many stages must coordinate to limit the aggregated rate of requests sent to the on-demand stages. We simulate an algorithm similar to the one described in §4.2.1, but add additional logic to allocate rates between these stages based on previous workload patterns.

Figure 4.7 shows that even though Original does not isolate C1 from C2, both the direct and indirect approaches provide stable throughput to C1 despite the change of C2. The indirect approach achieves lower throughput because it conservatively sets the rate limits and only probes for more when observing low resource utilization. An additional disadvantage of the indirect approach occurs with more nodes: each node introduces more information that must be shared with others, making the approach less scalable.

Since the direct approach is superior in this case, HBase-Maat adds scheduling points to solve the no scheduling problem (Maat-[d|?|?|?|?]).

4.3.2 Unknown Resource Usage

Problem: The RPC Handle stage *may* short-circuit reads and send responses to clients. Short-circuited reads could cause contention for I/O, yet this is not known when the request is scheduled. RPC Handle only sends responses when the network resource (i.e., the connection) is idle, so this do not interfere with scheduling.

We compare the direct splitting-stage approach ([d|d|?|?|?]) and the indirect speculative-execution approach ([d|i|?|?|?]) to handle short-circuited reads in the HBase RPC Handle stage. For the direct approach, we move the short-circuited read processing to the Data Xceive stage, and make RPC Handle non-blocking to enable independent scheduling. For the indirect approach, we track the workload pattern of each client; when CPU

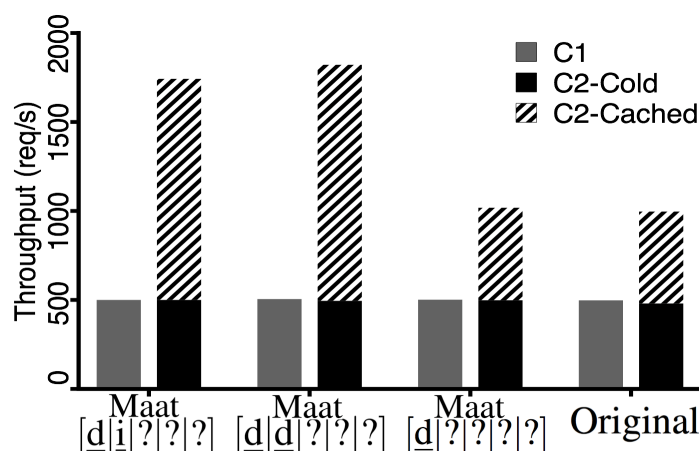


Figure 4.8: **Unknown Resource Usage Solutions.** Both C1 and C2 issue Gets on cold data, which incur 100 KB short-circuited reads at the RPC Handle stage. C2 also issues cached Gets that do not require I/O.

and network are idle, we speculatively execute requests from those clients who mostly issue CPU-intensive requests and abort these requests if they need I/O.

We compare these two approaches in a simulated standalone HBase node; this setting ensures that all HDFS reads at the RegionServer are short-circuited, thus isolating the effect of unknown resource usage. Figure 4.8 shows that both approaches achieve additional throughput for the cached Gets of C2, without reducing the throughput of C1 or C2’s cold-cache Gets. The indirect approach achieves slightly lower throughput than the direct approach because it aborts some requests, but this difference would decrease with faster CPUs since the cost of a failed speculation will be relatively lower.

HBase-Maat adopts the indirect speculative-execution approach (Maat-[d|i|?|?|?]) because its involves less change to the HBase system and remains highly efficient when CPU is relatively fast.

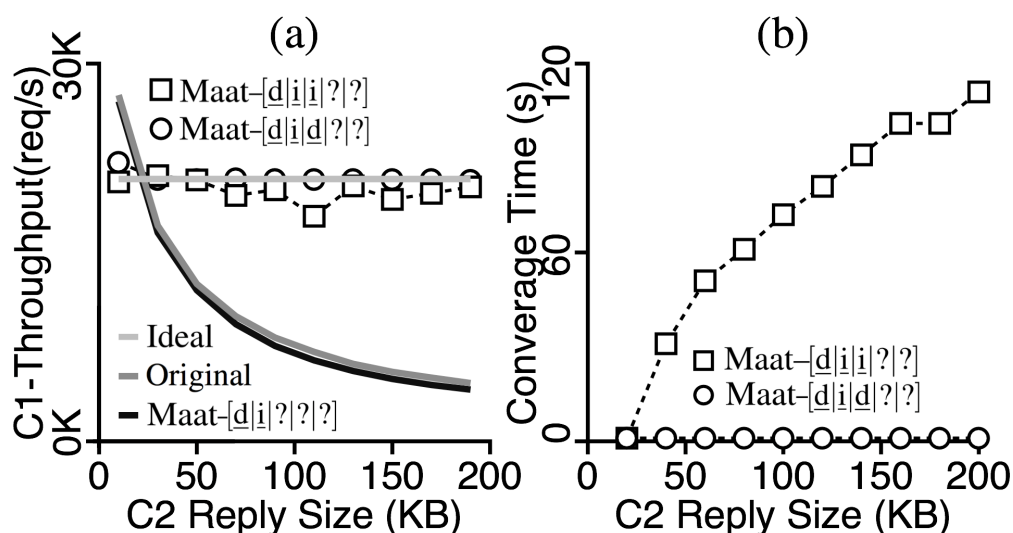


Figure 4.9: **Hidden Contention.** *C1 and C2 keep issuing RPCs with 1 KB message size. C1's response size remains 20 KB, while C2's response size varies from 10 KB to 200 KB. (a) shows the average throughput of C1; (b) shows the time the cluster takes to stabilize (client throughput fluctuation less than 5%) when C2's response size changes from 20 KB to various values.*

4.3.3 Hidden Contention

Problem: Both the RPC Handle and Data Xceive stage consume I/O; the RPC Read, RPC Handle, RPC Respond, Data Stream, and Data Xceive stage all contend for network; CPU resources are contended by many stages within the same node.

We investigate direct and indirect solutions to the hidden contention problem. The direct solution (Maat-[d|i|d|?|?]) combines scheduling points: all stages within the same physical node share the same scheduler. Note that we do not combine stages; different stages obtain their requests from the same scheduler.

In the indirect approach (Maat-[d|i|i|?|?]), a controller monitors resource usage and adjusts client weights at each stage. If stage S1 is excessively using resources on behalf of client C1, the weight of C1 is reduced across

all stages so that less C1 requests are issued to S1, forcing S1 to either use fewer resources or serve other clients.

Figure 4.9(a) shows that both approaches isolate C1 from C2's reply size change, versus Original and Maat- $[d|i|?|?|?]$ cannot provide isolation. Figure 4.9(b) shows when C2's workloads suddenly changes, the indirect approach takes a long time to converge to a fair state, depending on the amount of workload imbalance. The direct approach converges much more quickly.

In HBase-Maat, we adopt a hybrid of the above two approaches for ease of implementation. All stages within the RegionServer process share the same scheduler; the Data Xceive stage, which is in a different DataNode process, uses a separate scheduler and relies on the controller to coordinate with the RegionServer scheduler. We denote this as Maat- $[d|i|d|i|?|?]$, where d_i indicates a hybrid of direct and indirect.

4.3.4 Blocking

Problem: Three stages in the HBase RegionServer block on other stages: RPC Handle, Mem Flush and Log Sync. However, all Mem Flush or Log Sync requests follow the same data path (HDFS writes), and therefore do not cause a blocking problem. The RPC Handle stage, however, allows multiple data paths and is susceptible to the blocking problem. Indeed, the blocking problem of HBase occurs in production [16].

We compare one direct approach and two indirect approaches to solve the blocking problem in HBase. In the direct approach (Maat- $[d|i|d|i|d|?]$) RPC Handle is changed to be non-blocking, thus enabling independent scheduling at different stages.

In the first indirect approach (Maat- $[d|i|d|i|i^1|?]$), RPC Handle is modified to monitor the per-client congestion level in Data Xceive and avoids scheduling requests that may trigger excessive blocking. In the second indirect approach (Maat- $[d|i|d|i|i^2|?]$), RPC Handle threads are treated as

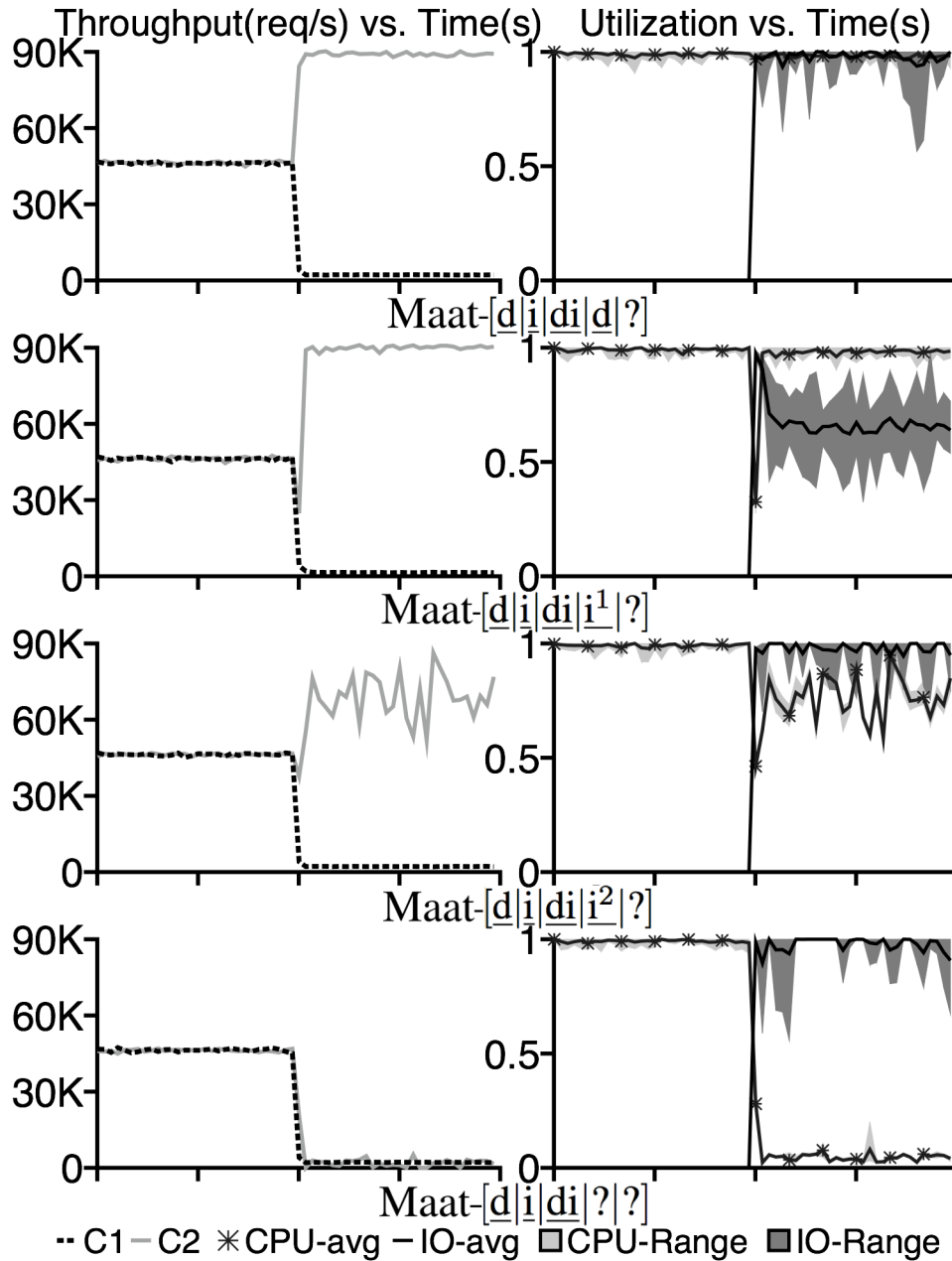


Figure 4.10: **Blocking.** Initially both C1 and C2 issue cached Gets. At time 60 C2 request uncached data, causing threads to block on I/O. The range of high and low CPU and I/O utilization is shown across the 7 slave nodes; also the average.

a resource and allocated between clients like CPU or I/O resources. This approach does not eliminate blocking, but prevents one client from occupying all RPC Handle threads and allows other clients to make progress. More upstream threads are needed with this approach to maintain the same utilization and more threads lead to less scheduling control and worse latency guarantees. The advantage is that upstream scheduling decisions are made locally, avoiding any control plane involvement.

Figure 4.10 shows that when C2 switches to an I/O intensive workload, all three solutions allow C1 to utilize CPU effectively and achieve high throughput, while Maat-[d|i|di|?|?] delivers very low throughput. Original behaves similarly to Maat-[d|i|di|?|?] so its result is omitted. The direct approach achieves the highest throughput and nearly perfect resource utilization. The I/O utilization of Maat-[d|i|di|i¹|?] fluctuates as the scheduler adjusts how aggressively it limits requests from C2 based on information from the Data Xceive stage. The CPU utilization of Maat-[d|i|di|i²|?] fluctuates as the scheduler allocates different numbers of RPC Handle threads to C1.

Figure 4.11 shows that for Maat-[d|i|di|i¹|?], both C1's throughput and the CPU utilization drop sharply as the probability of a wrong prediction increases. Similarly, for Maat-[d|i|di|i²|?], smaller number of RPC Handle threads leads to lower throughput and CPU utilization.

We incorporate the second indirect approach (Maat-[d|i|di|i²|?]) into HBase-Maat to solve the blocking problem, because it enables the scheduler to make local decisions and involves relatively few changes to HBase.

4.3.5 Ordering Constraints

Problem: The Log Append stage must process WAL entries in the order they are accepted and the writes it issues to HDFS must be processed in this order. Log Append enforces this by appending writes to the same file, which HDFS writes in sequence.

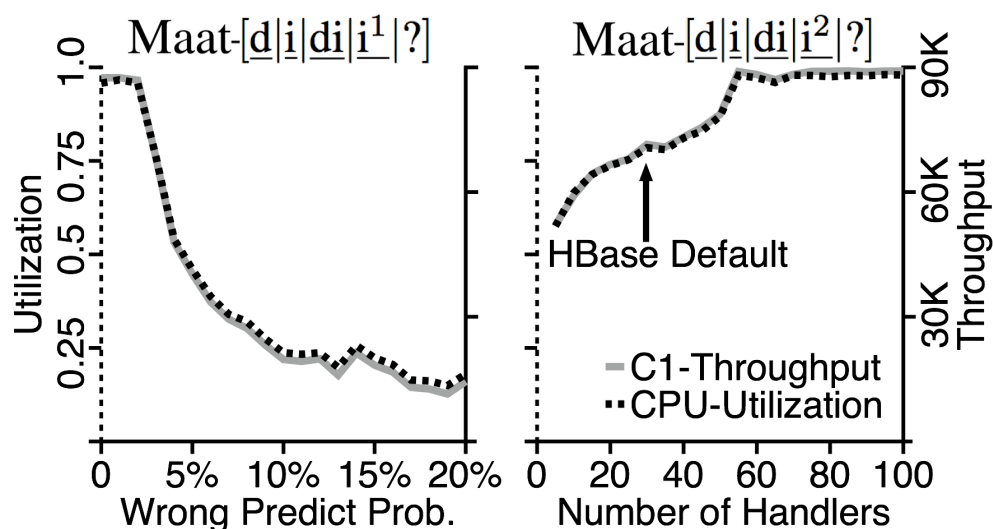


Figure 4.11: **Blocking.** *C1 issues cached Gets; C2 issues un-cached Gets that block on I/O. The left y-axis shows average CPU utilization; the right y-axis shows C1 throughput.*

We compare a direct and an indirect solution to the ordering constraint problem in HBase. For the direct approach (Maat-[d|i|d|i²|d]), we maintain one WAL per client, so there is no ordering constraints across clients.

For the indirect approach (Maat-[d|i|d|i²|i]), we schedule at the RPC Handle stage, 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 where queue size at each stage is relatively stable, the blocking time is roughly proportional to the number of downstream requests, and scheduling RPC Handle time indirectly schedules the WAL writes before passing them to the LOG Append stage. 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 Maat-[d|i|d|i²|i] with 10 or 30 RPC Handle threads.

Figure 4.12 shows that unlike in Original, where the throughput drop

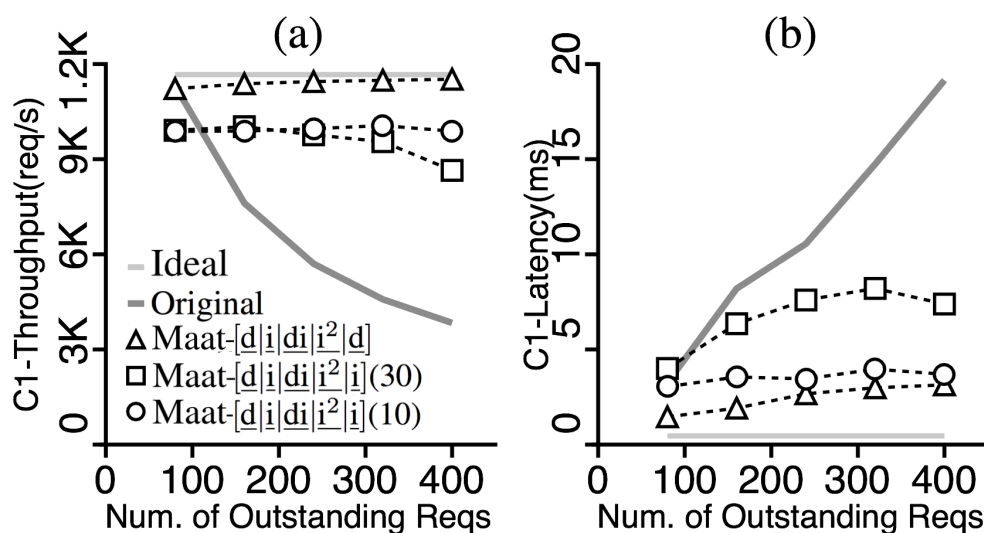


Figure 4.12: **Ordering Constraints.** Both C1 and C2 issue 1 KB Puts, resulting in 1 KB WAL appends. (a) shows the throughput of C1 when it issues a steady stream of requests using 80 concurrent threads. (b) shows the latency of C1 when it issues bursty requests every 1 second. 30 or 10 represents the number of RPC handler threads.

and latency increase are unbounded, the three solutions are able to limit C2's effect on C1. The separate-WAL approach (Maat-[d|i|d|i²|i]) achieves the best throughput and performance guarantee; the schedule-handle-time approach (Maat-[d|i|d|i²|i]) largely ensures isolation, but at the cost of lower throughput. With more RPC Handle threads, isolation becomes less effective and the latency guarantee degrades because C1 competes with more requests from C2 after they enter the RPC Handle stage.

For our final simulated system, we incorporate the indirect solution, resulting in HBase-Maat ([d|i|d|i²|i]).

4.4 HBase-Maat Implementation

To demonstrate that real storage systems can be implemented with the Maat principles, we now modify HBase according to the lessons we learned from the TAD simulations in §4.3. The HBase-Maat implementation gives us experience implementing Maat mechanisms in real systems and validates that the TAD simulations are excellent predictors of the real world.

4.4.1 Implementation Experience

Our implementation closely follows our final simulated solution: HBase-Maat ($(d_i d_i^2)_i$). To the original HBase, we add scheduling points to the Data Xceiver and Data Stream stages, and modify the other stages to allow pluggable schedulers. Different implementations of the local schedulers are possible; for our evaluation, we use DRF-based [47] weighted fair queuing as our local schedulers. RPC Handle time is treated as a resource in addition to the CPU, network, I/O and the namespace lock during scheduling. The scheduler only performs network resource scheduling when the server's bandwidth is the bottleneck; we anticipate incorporating global network bandwidth allocation [72] in the future.

To avoid hidden competition, within one RegionServer, the RPC Read, RPC Handle, RPC Respond, Data Stream and LOG Append stages all share the same multi-stage scheduler. Since it is difficult for the Data Xceiver stage, which resides in a separate Java process, to access the same scheduler, there is a separate Data Xceiver scheduler.

A centralized controller coordinates between the Data Xceiver scheduler and the RegionServer scheduler, as well as the schedulers on different nodes. The controller provides a Thrift [105] server interface and communicates with the scheduler using Protocol Buffers [113]. Each scheduler contains a thread that periodically sends status to the controller, retrieves the latest local weight allocations, and applies these allocations.

The original HBase/HDFS storage stack does not propagate the client identifier across the HBase/HDFS boundary; we add end-to-end client id propagation so that every scheduler can map a request back to its originating client.

One concern with multiple stages sharing the same scheduler is increased lock contention; even original HBase (in which only RPC Handle threads access a common scheduler) suffers from lock contention. We use the same solution as the original HBase: multiple scheduler instances, each protected by its own lock. While conceptually there is only one scheduler, in practice, an upstream stage places requests in a random scheduler instance, and each downstream thread is assigned a scheduler instance to pick requests from. We find that three instances of the multi-stage scheduler in each RegionServer works well.

Another concern with scheduling at every stage is additional context switches during request processing. For example, in the original HBase, each RPC Read thread monitors a set of connections and reads immediately when data is available. To enforce scheduling during RPC reading, though, a separate `ConnectionEnqueuer` thread performs monitoring and puts the connection with data available in queue. While the RPC Read threads pick connections in queue to read from, which introduces one extra context switch. To amortize this cost, we increase the scheduling unit and allow the RPC Read threads to read multiple RPC calls at once. Similarly, at the Data Xceive stage, we allow the scheduler to schedule multiple data packets at once to minimize the context switch cost.

4.4.2 TAD Validation

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.

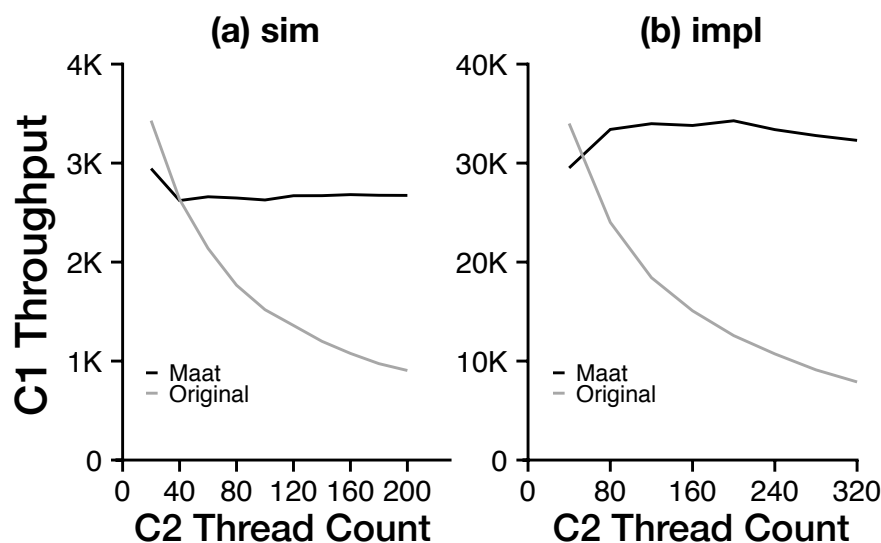


Figure 4.13: **HBase Implementation: No Scheduling.** We repeat the experiment in Figure 4.7. (a) shows the simulation results (same as Figure 4.7); (b) shows the results measured on a cluster.

One node hosts the HMaster, Namenode and Secondary Namenode; the other seven nodes host RegionServers and Datanodes.

We now re-run the simulated experiments in §4.3 and show side by side the simulated results and the results obtained from real implementation. We can see that both the problems and solutions in the implementation match the simulations of the TAD diagrams amazingly well; this accuracy holds across all five categories.

Figure 4.13 illustrates that the original implementation of HBase suffered from the no scheduling problem and as a result, the throughput of client C1 is significantly harmed when C2 issues more requests concurrently; further, Figure 4.13 shows that the solution of adding scheduling points at resource-intensive stages provides performance isolation in the real-world, as suggested in the simulation.

The unknown resource problem that exists in the original implemen-

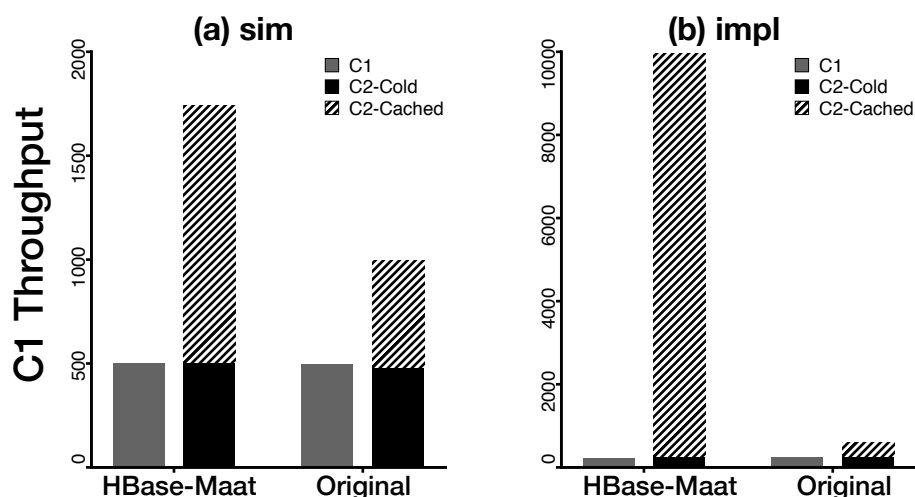


Figure 4.14: **HBase Implementation: Unknown Resource Usage.** We repeat the experiment in Figure 4.8. (a) shows the simulation results (same as Figure 4.8); (b) shows the results measured on a standalone node. A standalone node is used instead of a cluster to isolate the effect of short-circuit reads.

tation of HBase is shown in Figure 4.14: when client C2 requests in-cache data, HBase is not able to efficiently utilize the CPU. Our solution of speculatively executing requests to discover their resource consumption dramatically improves the throughput of C2 without harming C1. We see much higher relative throughput improvement for cached requests in implementation than in simulation because in our cluster the CPU is much faster (16 cores at 2.4 GHz with hyper-threading) than the simulated single-core 1 GHz CPU.

Figure 4.15 verifies that HBase suffers from hidden contention across multiple stages, which manifests when one client uses more resources in one stage (i.e., C2 uses more network). 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. The

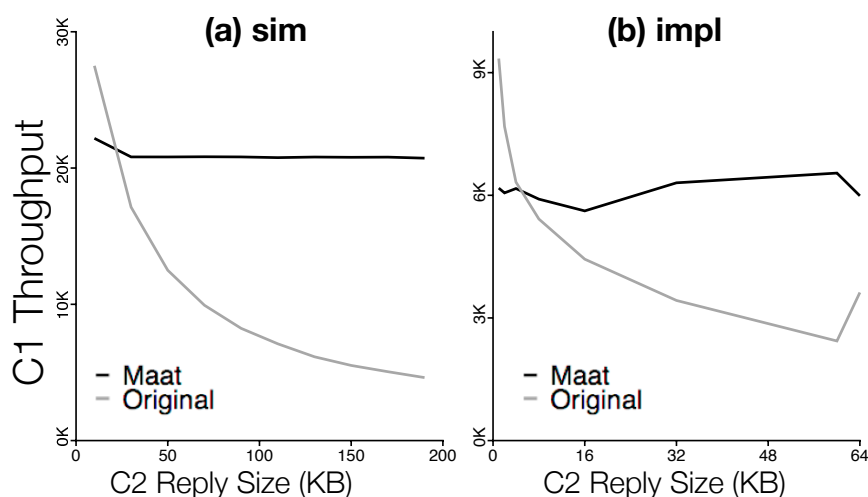


Figure 4.15: **HBase Implementation: Hidden Contention.** We repeat the experiment in Figure 4.9. (a) shows the simulation results (same as Figure 4.9); (b) shows the results measured on a cluster. To ensure that the network bandwidth is the bottleneck, the bandwidth between the nodes in the cluster is limited to 100 Mbps using *dummynet* [94].

solution suggested by our TAD simulations of producing a multi-stage scheduler again works well for the implementation.

The blocking problem that exists within HBase is illustrated in Figure 4.16. In the original HBase, when the workload of one client switches from CPU to I/O-intensive requests (C2 at time 60), both clients are harmed because not enough threads are available. In real implementation, unlike in simulation, the throughput of both clients increases to around 30 kops/s after a while, when the data C2 accesses are cached by the local file system in the page cache. The effect of the OS page cache is not captured in our simulation, but the general trend, that C1 is slowed down to the C2 throughput level, remain the same both in simulation and in real implementation. When disabling the page cache (not shown here) we observe that the throughput of both C1 and C2 remain low, as suggested by the simulation.

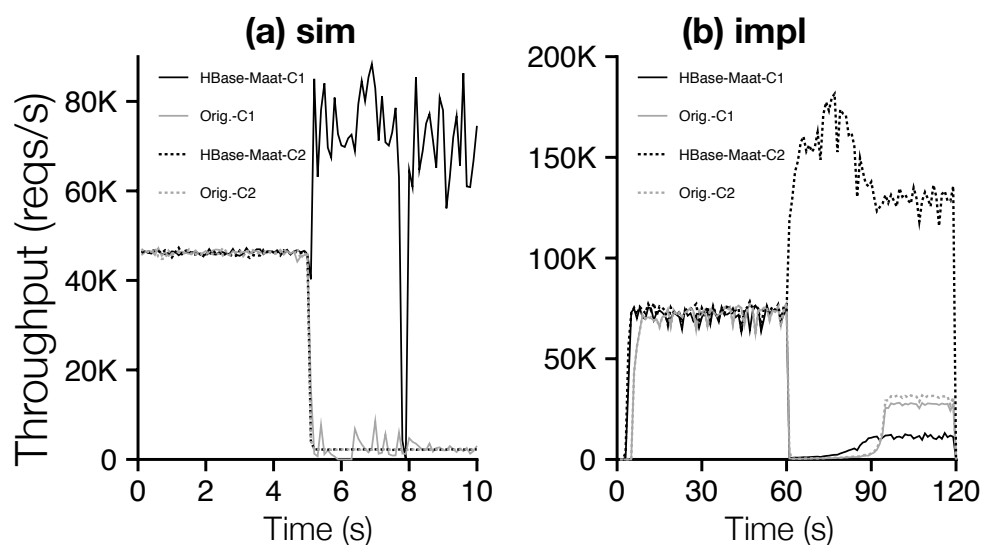


Figure 4.16: **HBase Implementation: Blocking.** We repeat the experiment in Figure 4.10. (a) shows the simulation results (same as Figure 4.10); (b) shows the results measured on a cluster.

Finally, HBase’s ordering problem is shown in Figure 4.17; when C2 writes more data, the throughput of C1 suffers. Again, this problem is fixed 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 suggested by the simulation.

The final performance of HBase-Maat for YCSB [35] is shown in Figure 4.18; in this workload, five different clients are each given a different weight and we would like weighted fairness. The original version of HBase was unable to provide weighted fairness across clients with different shares, instead delivering approximately equal throughput to each. The Figure shows that HBase-Maat enforces weighted fairness as desired.

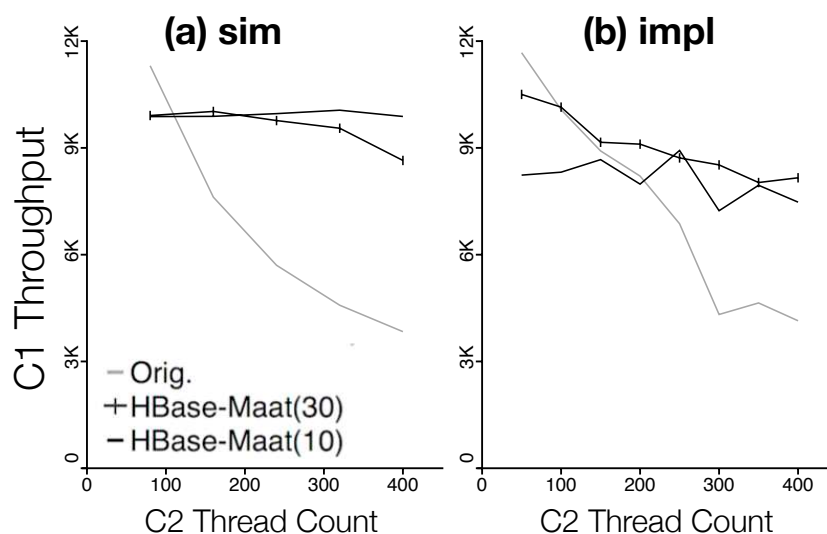


Figure 4.17: **HBase Implementation: Ordering Constraint.** We repeat the experiment in Figure 4.12. (a) shows the simulation results (same as Figure 4.12(a)); (b) shows the results measured on a cluster.

4.5 Analyzing Other TADs

Earlier (§3.4-3.6) we presented the TADs of MongoDB (Figure 3.3), Cassandra (Figure 3.4) and Riak (Figure 3.5). Here, we analyze each and discuss their scheduling problems.

4.5.1 MongoDB

MongoDB mostly resembles the traditional thread-based architecture, but its limit on active worker numbers and replication design are influenced by SEDA. MongoDB thus shares a lot of the scheduling problems of the thread-based architecture.

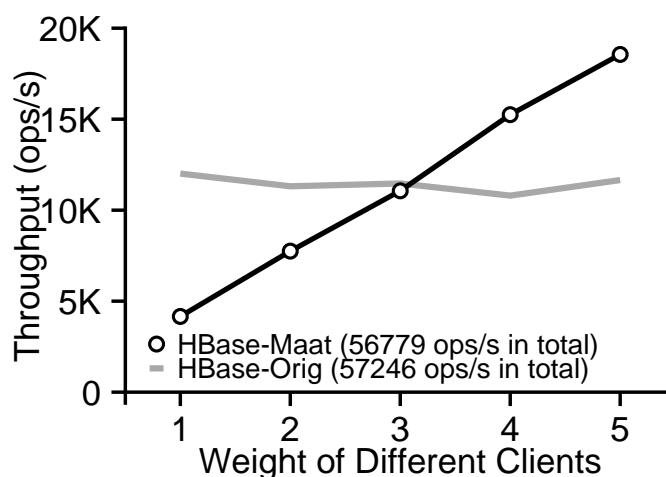


Figure 4.18: **Weighted Fairness on HBase.** 5 clients with different priorities run the YCSB workload on 8 nodes. The original version of HBase does not deliver weighted throughput; a version of HBase adhering to the Maat principles does.

4.5.1.1 Scheduling Challenges

Unknown Resource Usages: Unknown resource usage is the inherent scheduling problem associated with the traditional thread-based architecture, and MongoDB suffers from it heavily. The requests entering the Worker stage go through complex, highly variable execution paths until their completion. When the Worker threads read a request off the network, it does not know whether this request hits cache, requires replication, has to obtain exclusive locks, or needs I/O *within this stage*; it does not even have basis to make good predictions, making it extremely difficult to schedule at this point.

Hidden Contention: At the secondary node, multiple stages compete for the same resources, leading to the hidden contention problem. Most notably, the Worker stage and the Olog Writer stage compete for the database locks, causing reads on the secondary node to be delayed for an unbounded amount of time under heavy write load [17].

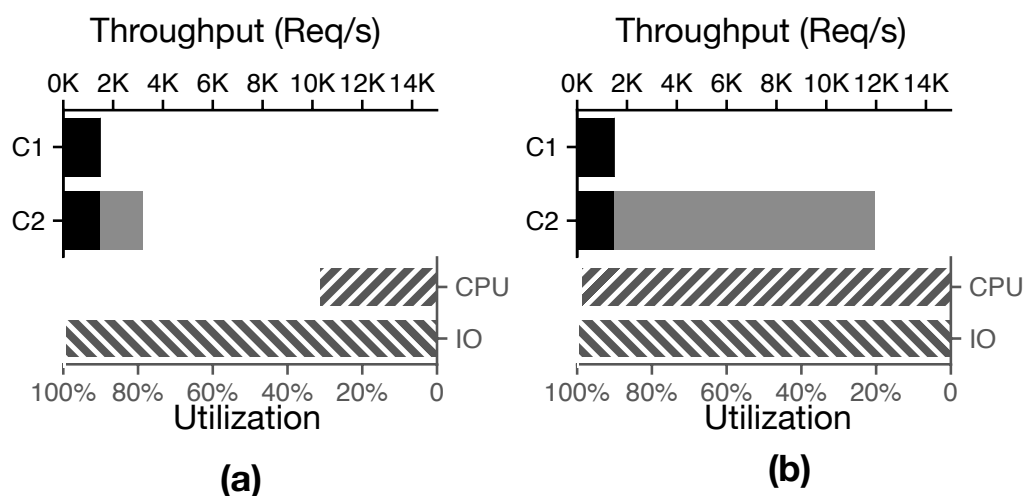


Figure 4.19: **MongoDB: Unknown Resource Usage.** Both C1 and C2 read cold data, which incur 1 MB I/O. C2 also issues cached Gets that do not require I/O. (a) shows the simulation of the original MongoDB; (b) shows the simulation when MongoDB processes the cached requests while CPU is idle.

Blocking: The Worker stage at the primary node blocks on the completion of data replication, and could run into the situation that it does not have enough active threads to execute requests that do not need replication, leading to the blocking problem.

4.5.1.2 TAD Simulation

We simulate an MongoDB cluster with 24 nodes that form 8 replication sets; each replication set consists of one primary node and two secondary nodes. Each node has one 1 GHz CPU and one disk with 100 MB bandwidth, and is connected via 1 Gbps network. In original MongoDB workers are scheduled to execute using a simple FIFO algorithm, we instead allow any scheduler and use DRF for our simulation.

Figure 4.19(a) shows the unknown resource usage problem on I/O resources. We could see that when C2 issues both cold- and hot-cache

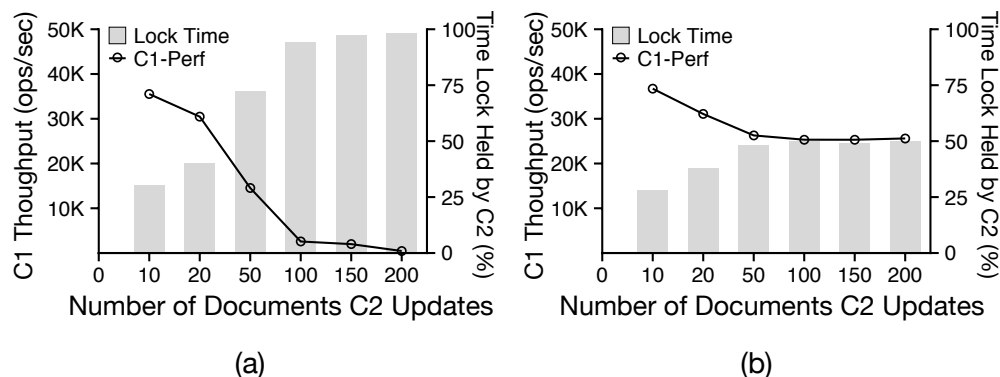


Figure 4.20: **MongoDB: Hidden Contention.** *C1 issues read requests to the secondary nodes; C2 issues write requests to the primary nodes that are replicated to the secondary nodes. (a) shows the simulation of MongoDB where separate DRF schedules at the Oplog Write stage and the Worker stage; (b) shows the simulation of MongoDB where a unified scheduler is used to schedule requests at multiple stages.*

requests, original MongoDB fails to utilize CPU effectively (less than 40% utilization) and provides low throughput to C2 cached requests. If MongoDB knows beforehand if a request needs I/O and schedules the cached requests when CPU is idle, both the CPU utilization and C2 throughput would be improved, as shown in Figure 4.19(b). However, as the Worker stage is so complex and performs many tasks, it is generally hard to predict the resource usage of a request.

Figure 4.20 shows the hidden contention on the database lock in the MongoDB secondary nodes. We can see that applying DRF separately at each stage does not guarantee C1 gets a fair share of the database lock time, as shown in Figure 4.20(a). Instead, when C2 updates more documents at the same time, which leads the Oplog Write stage to take more time to replicate the updates, it holds the lock for up to 98% of the time, and causes the throughput of C1 to drop from 35 Kops/s to less than 0.5 Kops/s. When using a unified scheduler, however, C2 only uses up

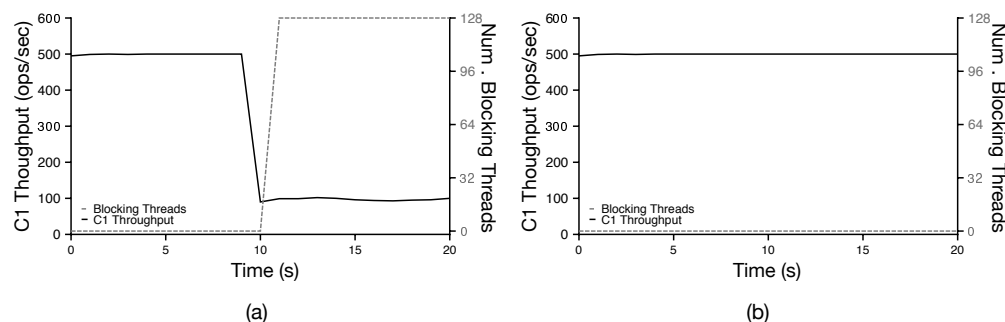


Figure 4.21: MongoDB: Blocking. *One replication set is simulated instead of 8. Both C1 and C2 issue requests to the primary node of the replication set; C1 read requests and C2 write requests. The writes of C2 are replicated to the secondary nodes. At second 10, the I/O bandwidth of the secondary nodes are reduced from 100 MB/s to 1 MB/s. (a) shows the simulation of original MongoDB, where the Worker threads block until the replication is done; (b) shows the simulation of MongoDB where the Worker threads do not block.*

to 50% of the database lock time, and C1’s throughput stabilizes as C2 updates more documents.

Figure 4.21 shows the Blocking problem of MongoDB. When the I/O bandwidth of the secondary nodes decreases, we expect the throughput of C2 to decrease since it relies on replication but the throughput of C1 to remain stable. However, as we can see from Figure 4.21(a), the throughput of C1 suffers because all 128 Worker threads are blocking and no threads are available to process the read requests. After changing the Worker stage to be non-blocking, as shown in Figure 4.21(b), the throughput of C1 is not affected.

4.5.1.3 Lessons

MongoDB resembles the traditional thread-per-request architecture and thus suffers from unknown resource usage. The complex execution path within the Worker stage makes indirect solutions to this problem extremely challenging to implement. We expect that altering MongoDB to com-

ply with the Maat principles will be difficult and may require substantial structural changes.

One may consider alternative ways of scheduling on MongoDB, e.g., intercepting resource accesses and suspending the accessing threads as needed. Libra [102] used this approach for I/O resource scheduling. However, it dictates the scheduling granularity (each resource access), imposes high overhead when scheduling fast resources such as CPU, and does not address the scheduling problems that requires scheduling at higher level (blocking, ordering constraints).

4.5.2 Cassandra

Cassandra closely follows the standard SEDA architecture, where all activities are managed in controlled stages. However, schedulability does not automatically follow. Having too many stages with the same resource pattern leads to hidden contention and the "inability to balance reads/writes/compaction/flushing" [14]; likewise, CPU- and I/O-intensive operations in the same stage leads to unknown resource usage. More specifically, Cassandra has the following scheduling problems.

4.5.2.1 Scheduling Problems

Unknown Resource Usages: The database processing stages have no knowledge on whether a request needs I/O when picking up requests for execution, leading to the unknown resource usage problem.

Hidden Contention: Cassandra has more than 10 database processing stages (Read, Mutation, View-Mutation, Counter-Mutation etc.); all of them compete for the same CPU and I/O resources. In addition, four different network-related stages compete for the network resources. Those competitions lead to the hidden contention problem. The Cassandra de-

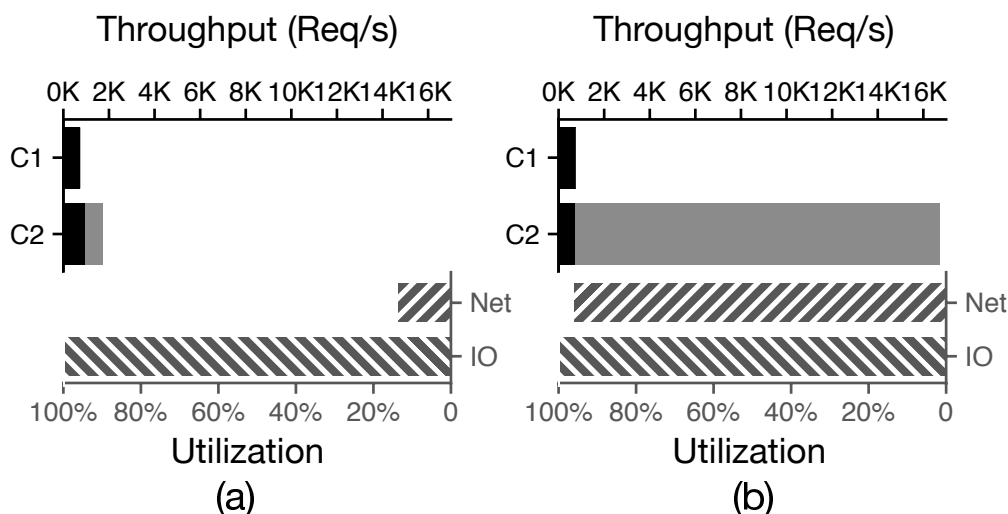


Figure 4.22: **Cassandra: Unknown Resource Usage.** Both C1 and C2 read cold data, which incurs 100 KB I/O. C2 also issues reads repeatedly on a single key, which does not require I/O. (a) shows simulation of the original Cassandra; (b) shows the simulation of an improved version of Cassandra, which speculatively executes cached requests when network and CPU are idle.

velopers also realize that such a design causes "inability to balance reads/writes/compaction/flushing" [14].

Blocking: The C-ReqHandle stage blocks on the request completion event, and could run into the situation where it has no additional threads to process more requests that can be finished without waiting for the blocking requests, leading to the blocking problem.

4.5.2.2 TAD Simulation

Based on the TAD of Cassandra shown in Figure 3.4, We simulate a Cassandra cluster with 3 nodes. Each node has one 1 GHz CPU and one 100 MB/s disk, and is connected via 1 Gbps network.

Figure 4.22 shows the unknown resource usage problem of Cassandra on I/O resources. We could see that when C2 issues both cold- and

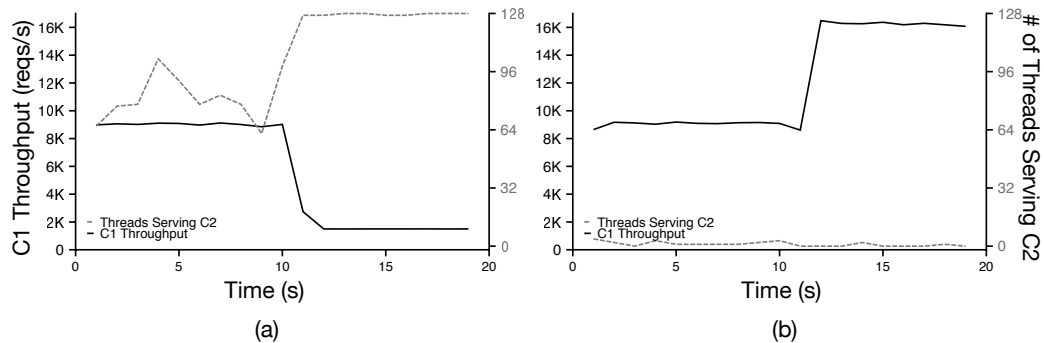


Figure 4.23: Cassandra: Blocking. Initially both C1 and C2 issue issue cached requests. At time 10 C2 starts to request uncached data, causing its processing threads to block for a longer time. The left y-axis shows the number of C-ReqHandle threads currently serving C2 (128 C-ReqHandle threads in total). (a) shows the simulation of original Cassandra; (b) shows the simulation of an improved version of Cassandra, where the C-ReqHandle stage does not block on the processing stages.

hot-cache requests, original Cassandra, without the exact knowledge of resource usage for each request, fails to utilize network effectively (less than 20% utilization) and provides low throughput to C2 cached requests. However, by speculatively executing requests when network and CPU are idle, Cassandra could fully utilize network and provide much higher throughput (8x improvement) for C2.

Figure 4.23 shows the blocking problem of Cassandra. When C2 switches from an CPU-intensive workload to an I/O-intensive one, we expect the throughput of C1 to increase since more CPU resources become available. However, as we can see from Figure 4.23(a), the throughput of C1 declines abruptly. All C-ReqHandle threads are blocked waiting for the processing of C2 requests to complete, which takes a long time since it requires I/O; no threads are left to serve C1. When we change the C-ReqHandle stage to be non-blocking, as shown in Figure 4.23(b), the throughput of C1 improves as expected after C2's workload change; the number of serving C-ReqHandle threads remains low because C-ReqHandle stage is not

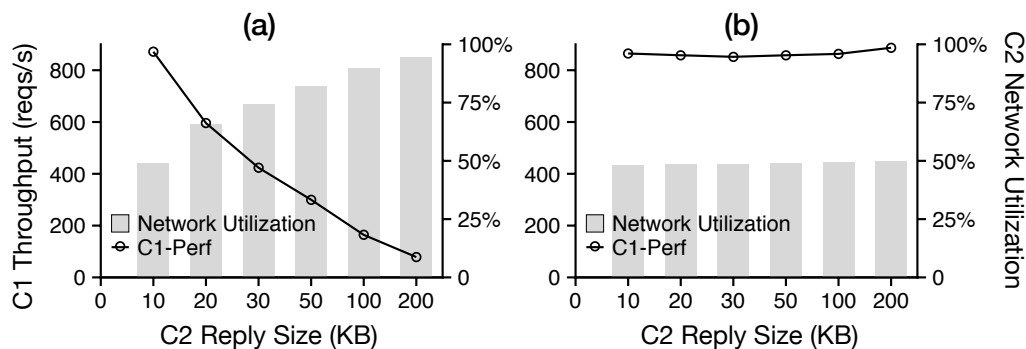


Figure 4.24: **Cassandra: Hidden Contention.** *C1 and C2 keep issuing requests with 1 KB message size. C1's response size remains 10 KB, while C2's response size varies from 10 KB to 200 KB. (a) shows the simulation of original Cassandra with fair scheduler at each stage; (b) show the simulation of an improved version of Cassandra, where a unified scheduler is used to schedule requests at multiple stages.*

the bottleneck and it does not have to wait for other stages.

Figure 4.24 shows the hidden contention on network resources in Cassandra. We can see that attempting fair scheduling at each stage does not ensure fair sharing globally. Instead, as C2 increases its response size, it unfairly consumes as much as 95% of the network resource, leading to declined C1 performance. When using a unified scheduler, however, all contentions on the network are explicitly managed since the scheduler has a global view of network usage. As a result, C2 can only use its fair share of network, and C1's throughput is not affected by C2's behavior.

4.5.2.3 Lessons

Simply following the SEDA design principle does not guarantee schedulability. 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 [14].

4.5.3 Riak

Riak is another system that closely follows SEDA. However, instead of using controlled stages, Riak heavily uses on-demand stages and relies on the underlying Erlang virtual machine to schedule the potentially large number of threads. From the TAD of Riak we can identify the following scheduling problems.

4.5.3.1 Scheduling Problems

No Scheduling: Riak has no control on how the Req In-Out stage and the Req Process stage access the CPU and network resources. The underlying Erlang VM, which is in charge of scheduling these processes, is oblivious about the storage system's scheduling goals and cannot be used as a general mechanism to provide flexible, application-specific scheduling policies.

Unknown Resource Usage: Depending on whether the Req Process thread and Cmd Handle thread locate on the same physical node, communications between them may consume network resources. However, Erlang uses a completely transparent IPC mechanism which does not distinguish local and remote communication. The scheduler can not tell whether a request uses network resource or not, even after the request execution (which precludes speculative execution as an indirect solution).

Hidden Contention: In Riak, all stages may compete for network resources; the Req In-Out stage and the Req Process stage compete for the CPU resource; and the Cmd Handle stages from different partitions within

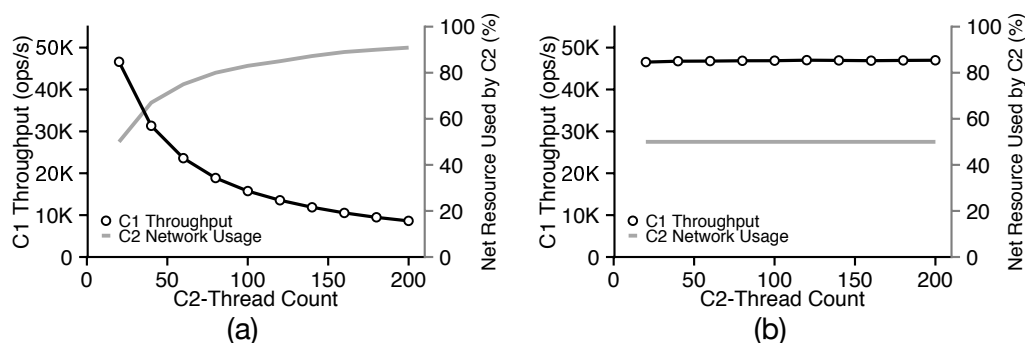


Figure 4.25: **Riak: No Scheduling.** Both C1 and C2 issue random read requests that are cached in memory (so that only the network resource is in contention). (a) shows the simulation of original Riak; (b) shows the simulation of a modified Riak version where both the Req In-Out and Process stages are controlled stages using DRF schedulers.

the same node compete for I/O resources. None of the contentions are explicitly managed.

4.5.3.2 TAD Simulation

Based on the TAD of Riak shown in Figure 3.5, we simulate a Riak cluster with 4 physical nodes. Each node has one 1 GHz CPU and one 100 MB/s disk, and hosts 4 data partitions (i.e., has 4 Cmd Handle stages). Nodes are connected via 800 Mbps network.

Figure 4.25(a) shows the no scheduling problem of Riak. We can see that original Riak fails to provide isolation: when C2 issues requests with more threads, it consumes more network resources and the throughput of C1 suffers greatly. After adding scheduling points at the previous on-demand Req In-Out and Process stages, as shown in Figure 4.25(b), network resources are shared fairly between C1 and C2, and the throughput of C1 is not affected by C2.

Figure 4.26(a) shows the unknown resource usage problem of Riak. Without the knowledge of the network resource usage, the scheduler can-

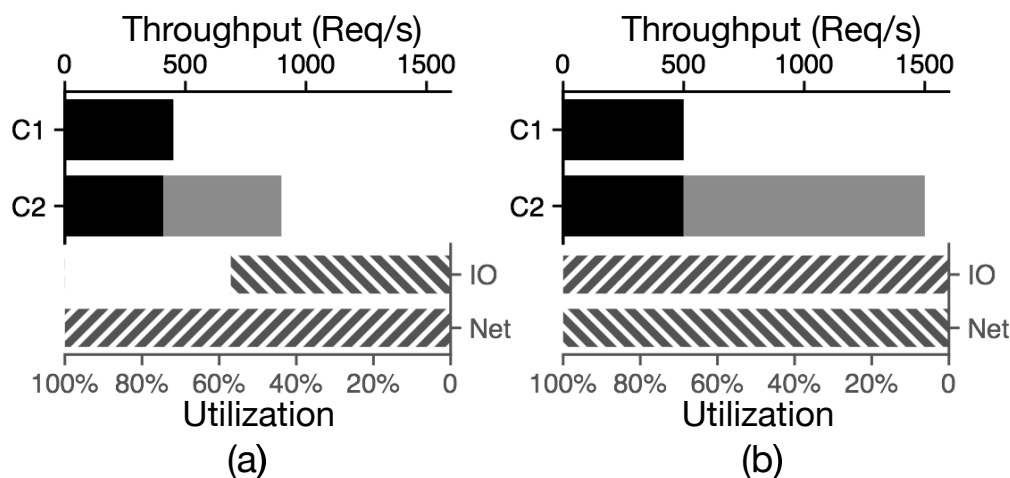


Figure 4.26: **Riak: Unknown Resource Usage.** *C1 issues requests to one node but the data it requests reside on another node (thus requires network communication between these two nodes); C2 issues requests that access data at both the local and remote nodes. (a) shows the simulation of Riak where the Process and Cmd Handle stage schedule the network resources without knowing whether a request needs it; (b) shows the simulation of Riak where the Process and Cmd Handle stages are split so that network communications are handled in separate stages that have knowledge of the network resource usage.*

not effectively utilize the I/O resource by scheduling requests that do not require network transfers, leading to lower throughput of C2. When separate network handling stages are added, which has exact knowledge of the network usage, though, requests can be scheduled based on their resource usage patterns. As a result, the I/O resource is fully utilized and the throughput of C2 improves as more requests that access local data are served, as shown in Figure 4.26(b).

Figure 4.27(a) shows the hidden contention on I/O resources among the Cmd Handle stage in Riak. When C2 issues larger I/O requests, we can see that it uses more I/O resources (up to 70%) and causes the throughput of C1 to decrease. Even though the scheduler at each stage attempts to allocate I/O resource fairly, the contention between these stages is un-

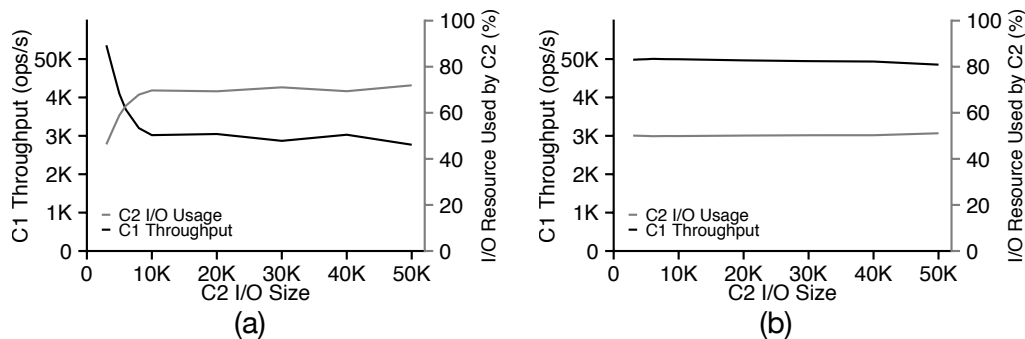


Figure 4.27: Riak: Hidden Contention. *C1 keeps issuing 5 KB read requests; the request size C2 issues vary. The Zpif distribution governs which partitions the requests go to for both C1 and C2. (a) shows the simulation of Riak where each Cmd Handle stage has a separate DRF scheduler; (b) shows the simulation of Riak where all stages in one physical node DRF schedulers.*

regulated and causes C2 to receive more resources when it issues larger requests. When a unified scheduler is used, as shown in Figure 4.27(b), C2 always receives a fair share of the I/O resources and does not affect the performance of C1.

4.5.3.3 Lessons

Riak relies heavily on light-weighted processes and transparent IPC provided by the Erlang virtual machine, which makes resource management implicit. Creating a new Erlang process may have low overhead, creating them on-demand leads to the no scheduling problem. Similarly, transparent IPC hides important network usage information, making scheduling the network difficult. To make Riak comply with Maat principles, one must either explicitly manage the above mechanisms, or change Erlang VM to allow scheduling policies to be passed from Riak to the VM level.

	N	U	C	B	O
HBase [46]	✘	✘	✘	✘	✘
MongoDB [34]		✘	✘	✘	
Cassandra [73]		✘	✘	✘	
Riak [71]	✘	✘	✘		

Table 4.5: **Different Storage Systems Presents Different Scheduling Challenges.** ✘:have the corresponding problem

4.5.4 Discussions

Table 4.5 presents a summary of the systems we study and their scheduling problems; we now discuss some general findings across systems.

First, some problems are more common than others; for example, the hidden contention on CPU is hard to avoid as most stages use CPU. We observe the unknown resource usage and hidden contention problem on every system we study, but the ordering constraint problem only in HBase.

Second, fixing one problem might introduce other problems in the system. For example, Riak does not have a blocking problem. Even though its Req In-Out and Process stages block on downstream stages, these stages are on-demand and can always spawn a new thread to handle new requests; no requests would be denied service in Riak due to a lack of active (not blocked) threads. However, if one changes the Req In-Out and Process stages to be bounded to solve the no scheduling problem of Riak, one introduces the blocking problem to Riak because now all threads in the bounded stages may be blocked, leaving no threads available to process non-blocking requests.

Finally, even though different systems might possess the same problem, the difficulty of fixing that problem 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 the same problem directly in MongoDB, however, requires a major re-structuring of the Worker

stage to account for the the complex execution path within that stage. Our TADs are effective in identifying the problems, but do not give many indications on how difficult solving the problems would be; systematically reasoning about such difficulties is an interesting direction to extend the thread architecture diagrams.

4.6 Conclusions

We have presented Maat, an approach to building schedulable distributed storage systems. By using direct or indirect approaches, Maat enables systems to overcome the five fundamental problems of schedulability and deliver fairness, isolation, and other important performance properties. We show both via simulation and implementation how to apply the Maat principle to enable scheduling in the HDFS/HBase storage stack. We also discuss the scheduling problems of MongoDB, Cassandra and Riak using the Maat principle.

5

Related Work

In this chapter, we discuss various research efforts and systems that are related to this dissertation. We start by discussing works related to local storage system scheduling (§5.1). We then discuss how previous research proposed to realize scheduling in distributed storage systems (§5.2).

5.1 Scheduling in Local Storage System

Deciding which I/O request to schedule, and when, has long been a core aspect of the local storage stack in the operating system, and there is a rich history in the system community to improve it [11, 13, 22, 27, 28, 29, 31, 38, 43, 44, 45, 54]. Each of these approaches has improved different aspects of I/O scheduling. In this section, we discuss how these works on local storage stack scheduling are related to our proposed split-level I/O scheduling in five different aspects: multi-layer scheduling, cause-mapping and tagging, software-defined storage, exposing file-system mechanisms, and I/O scheduling algorithms.

Multi-Layer Scheduling: Split-level I/O advocates for scheduling at multiple layers; a number of works also argue that efficient I/O scheduling requires coordination at multiple layers in the storage stack [101, 114, 118, 126]. Riska *et al.* [93] evaluated the effectiveness of optimizations at various layers of the I/O path, and found that superior performance is

yielded by combining optimizations at various layers. Redline [126] tries to avoid system unresponsiveness during `fsync` by scheduling at both the buffer cache level and the block level. Argon [114] combines mechanisms at different layers to achieve performance insulation. However, compared to these ad-hoc approaches that require re-engineering of the whole storage stack, the split I/O framework provides a systematic way for schedulers to plug in logic at different layers of the storage stack while still maintaining modularity.

Cause Mapping and Tagging: The need for correctly accounting resource consumption to the responsible entities arises in different contexts in addition to the tagging needed in the split-level I/O scheduling framework. Banga *et al.* [22] found that kernel consumes resources on behalf of applications, causing difficulty in scheduling. The hypervisor may also do work on behalf of a virtual machine, making it difficult to isolate performance [58]. We identify the same problem in I/O scheduling, and propose tagging as a general solution. Both Differentiated Storage Services (DSS) [88] and IOFlow [109] also tag data across layers. DSS tags the type of data, IOFlow tags the type and cause, and split scheduling tags with a set of causes. Multi-causes tags that represent many-to-many mappings between causes and operations are important in local storage, as metadata and journal I/Os frequently have multiple causes.

Software-Defined Storage Stack: In the spirit of moving toward a more software-defined storage (SDS) stack, the split-level framework exposes knowledge and control at different layers to a centralized entity, the scheduler. The IOFlow [109] stack is similar to split scheduling in this regard; both tag I/O across layers and have a central controller.

IOFlow, however, operates at the distributed level; the lowest IOFlow level is an SMB server that resides above a local file system. IOFlow does not address the core file-system issues, such as write delegation or ordering requirements, and thus likely has the same disadvantages as system-

call scheduling. We believe that the problems introduced by the local file systems, which we identify and solve in this dissertation, are inherent to any storage stack. We argue any complete SDS solutions would need to solve them and thus our approach is complementary. Combining IOFlow with split scheduling, for example, could be very useful: flows could be tracked through hypervisor, network, and local-storage layers.

Shue *et al.* [102] provision I/O resources in a key-value store (Libra) by co-designing the application and I/O scheduler; however, they noted that “OS-level effects due to filesystem operations [...] are beyond Libra's reach”; building such applications with the split framework should provide more control.

Exposing File-System Mechanisms: Split-level scheduling requires file systems to expose certain mechanisms (journaling, delayed allocation, etc.) to the framework by properly tagging them as proxies. Others have also found that exposing file-system information is helpful [44, 90, 125]. For example, in Featherstitch [44], file-system ordering requirements are exposed to the outside as dependency rules so that the kernel can make informed decisions about writeback.

I/O Scheduling Algorithms: Split-level I/O scheduling framework does not stipulate the scheduling algorithm one uses; it only provides more information and control to any scheduler one implements. Different I/O scheduling algorithms have been proposed to improve different aspects of I/O scheduling: to better incorporate rotational-awareness [63, 65, 100], to better support different storage devices [68, 89], or to provide better QoS guarantees [56, 77, 92]. All these techniques are complementary to our work and can be incorporated into our framework as new schedulers. These schedulers would benefit from the additional information and control the split framework provides.

5.2 Scheduling in Distributed Storage Systems

Distributed storage systems that host data from many clients are widely adopted [43, 46, 71, 73]. However, most systems today provides very weak performance guarantees such as isolation, fairness, or bounded latency, if at all. Various research efforts have been made to enable scheduling and performance guarantees in these systems, with different extent of success. However, to the best of our knowledge, no study has looked at the *schedulability* of these systems in a systematic way.

Several systems focus on coarse-grained resource allocation on a per-VM basis rather than on per application-level request. For example, mClock [57] and PRADA [55] support proportional-share fairness on storage server bandwidth allocation across client virtual machines. IOFlow [109] uses a logically centralized control plane to enable scheduling policies at the VM level. PriorityMeister [130] employs a combination of priorities and rate limits to provide tail latency QoS on NFS for each virtual machine. These systems usually schedule low-level I/O resources (block level or file system level) and need not deal with the complex internal structure and mechanisms of replicated, fault-tolerant, distributed storage systems with rich data semantics such as HBase or Cassandra. Performance guarantee at the VM-level also do not meet the need for finer-grained control, where request-level scheduling is needed.

More recently, Pisces [103] investigates how to perform request scheduling in distributed key-value store systems. Pisces provides global coordination mechanisms to adjust the scheduling policy within each node to achieve the overall scheduling goal. However, it avoids the complexities of enforcing the scheduling policy in complex thread architectures as they assume an unrealistically simple storage server design with only in-memory operations. Libra [102] is complementary to Pisces as it provides a local I/O scheduling framework that works at each storage node to enforce higher-level policies. Libra is implemented as a wrapper layer

on top of POSIX file systems, plus modifications to levelDB to pass the client identity. Libra works by delaying the threads calling the file system APIs, thus is only suitable to schedule I/O resources. However, different resources may become bottlenecks in a storage system [80], so they all need to be monitored and managed. Otherwise the system would run into the no scheduling problem, as we identified in this dissertation.

Cake [117] enforces SLOs by inserting schedulers on the RPC Handle and Data Xceive stages and using a centralized controller to adjust the schedulers. However, Cake does not cover the resource consumptions in other stages, such as RPC Read or RPC Respond. Argus [128] schedules cache and I/O resources in HBase. It partitions cache and uses a local scheduler in each RegionServer's RPC Handle to schedule I/O resources. A centralized scheduler co-located with HMaster pushes resource reservation policy. Neither system puts thoughts on where the scheduling points should be; they simply place the scheduling logic at the most convenient places. Unfortunately, such choices lead to ineffective scheduling, including the no scheduling problem as some resource-intensive stages are not managed.

Retro [80] applies rate limits to multiple scheduling points to emulate different scheduling policies at the Hadoop stack. Retro does not have scheduling points at every resource intensive stage, instead it uses the indirect approach we described in §4.2.1 to limit the number of requests sent to the un-scheduled stages. In Retro, all schedulers are treated the same way: if one client uses too much resource at one stage, all other stages across all nodes would be throttled, including those requests that do not use that type of resource at all, leading to inefficiency. Retro thus cannot solve the scheduling problems that involves multiple stages interacting with each other, such as blocking.

Pulsar [19] takes a different approach in scheduling. Pulsar uses a centralized controller to determine local scheduling policy, and enforces

the local scheduling policy within the hypervisor on the client side. For key-value storage, Pulsar estimated the cost of Puts and Gets based on a single-dimension cost function, and throttles a client if its requests to the storage server cost more than its fair share. This approach requires the storage server to trust the clients to schedule themselves, which is not always possible, especially in a cloud setting. The client side cost estimation could also be far off due to caching and other internal mechanisms in the storage server.

There are also many cluster scheduling solutions [27, 48, 51, 52, 60] that allocate per-node CPU and memory to schedule batch jobs and VMs. While they may provide useful insights, due to the unique challenges present in scheduling I/O resources, they are not directly applicable for scheduling in storage systems.

6

Conclusions and Future Work

In this chapter, we summarize our studies (§6.1) and list some of our lessons learned (§6.2). Finally, we describe our plans for future work (§6.3) and conclude (§6.4).

6.1 Summary

This dissertation showed how the scheduling in current local and distributed storage systems is inadequate, and proposed new frameworks to enable scheduling in these systems.

For the local storage stack, we have shown that single-layer schedulers operating at either the block level or system-call level fail to support common goals due to a lack of coordination with other layers. While our experiments indicate that simple layering must be abandoned, we need not sacrifice modularity. In our split framework, the scheduler operates across all layers, but is still abstracted behind a collection of handlers. This approach is relatively clean, and enables pluggable scheduling. Supporting a new scheduling goal simply involves writing a new scheduler plug-in, not re-engineering the entire storage system.

For distributed storage systems, we first introduced thread architecture diagrams as a tool to assess the schedulability of a system and analyze its scheduling problems. We used TAD to analyze popular stor-

age systems such as HBase, Cassandra and MongoDB. We then presented Maat, an approach to building schedulable distributed storage systems. By using direct or indirect approaches, Maat enables systems to overcome the five fundamental problems of schedulability and deliver fairness, isolation, and other important performance properties. We showed both via simulation and implementation how to apply the Maat principle to enable scheduling in the HDFS/HBase storage stack. We also discussed how to use the Maat principle to discover and fix the scheduling problems in MongoDB, Cassandra and Riak.

6.2 Lessons Learned

We now describe some general lessons on scheduling we learned while working on this dissertation.

Scheduling can be modeled at the request flow level, abstracting out most implementation details: One can understand the scheduling problem at a system by simply looking at how requests flow through various components of the system and consume resources, and how schedulers are placed at key points in the system to shape the flow. At this level, visualization tools (e.g., Figure 2.2 for the local storage stack, or TADs for distributed storage systems) can be introduced to achieve quick insights, and simulation can be utilized to study system behavior in detail without having to deal with cumbersome implementation details.

Where to place scheduling logic matters: Traditionally, few thoughts have been given to where to place the scheduling logic in complex storage systems. Due to the interactions between different components of the system, which may have unexpected effects on scheduling, it might be hard to enforce scheduling disciplines at certain points in the system. For example, one part of the system may be blocking on the progress of another part, or imposes artificial ordering limits. It is thus important to

understand these interactions and how they effect scheduling; TAD is one tool to facilitate such understanding. Equipped with the understanding of the inner works of the storage system, one can make more informed decisions on where to place schedulers to most effectively achieve one's scheduling goals.

Proper framework support is a prerequisite for effective scheduling: Just having sophisticated scheduling algorithms is not enough to achieve the desired scheduling goals. A scheduler needs both the *information* to make the right scheduling decisions and the *control* to implement these decisions. However, as we have shown in this dissertation, inadequate framework support (e.g., unknown resource usage or ordering constraint) often leaves the scheduler unable to perform effective scheduling.

Vertical integration is necessary for multi-layer storage stacks: Modern storage systems are complex and composed of multiple layers; scheduling at one single layer cannot achieve end-to-end performance goals. Both the split-level framework and the Maat approach deploy schedulers at multiple points to enforce scheduling; split scheduling in particular does so in a modular and systematic way, without requiring individual schedulers to re-engineer the complex storage stack. We hope that our work will inspire future vertical integration in storage stacks. RCP [69] is one such effort that further explores how to enable I/O prioritization by integrating multiple layers at the I/O path.

Schedulability should be made a first-class citizen in system design: Retrofitting scheduling control into existing systems is difficult; various architectural and structural problems can prevent effective scheduling from being realized. When scheduling is added as an afterthought to the system, these problems are almost unavoidable. One can work around the structural problems by adding information flows and feedback control. However, these indirect approaches only provide approximate scheduling controls; they also add overheads to the system control plane and

severely limit the scalability of the system. Directly solving these problems, on the other hand, often involves significant re-structuring of the system, which implies a large amount of engineering effort.

To avoid the lengthy and often painful process of making a problematic system schedulable, schedulability should be treated as a first-class citizen in the system design process; architectural choices that may introduce scheduling problems should be thoroughly compared with their alternatives. Depending on the system requirement, sometimes schedulability need to be compromised to achieve other even more desirable features of the system; however, such compromises have to be made as conscious decisions other than being mere coincidences. Such schedulability evaluation and optimization at the system design phase would save the system designers and developers a great deal of trouble later when they realize various scheduling policies and provide performance guarantees.

6.3 Future Work

In the era of cloud computing and big data, more and more data are stored in the shared infrastructure in large data centers. Scheduling accesses to these data in an efficient and scalable way has become more compelling an issue than ever. We see many opportunities in optimizing the current storage stack and providing the much needed performance guarantees. In this section, we discuss several directions to extend the work in this dissertation and build storage systems that better meet the requirement in modern data centers.

6.3.1 Scalable Scheduling Architecture

Today's data centers have tens of thousands of nodes and millions of cores; any scheduling framework operating in these data centers need the ability to scale to such magnitudes.

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. To achieve scalability, one needs to place as much work as possible at the local schedulers, while minimizing the control plane involvement for information collection, computation and coordination.

However, as we discussed earlier in this dissertation, a storage system may have various architectural problems that prevent the local schedulers from being effective. Additional information and coordination across the local schedulers can be utilized to circumvent these scheduling obstacles, but at the cost of scalability. For example, two local schedulers competing for the same resource may lead to unfairness, as shown in §4.2.3. In this case, simply allocating local weights at the two schedulers is not sufficient, the control plane has to be involved to coordinate resource consumption between these schedulers, creating extra global work. We thus believe that the architecture of a storage system ultimately limits how scalable its scheduling can be, and such limits are independent of how the scheduling is implemented.

Denote the average workload change time in a storage system by w , and the average time the distributed scheduler needs to respond to the workload (collecting the information about the workload, devising a new scheduling plan, enforcing the plan at the local schedulers and stabilizing) as s ; one can quantify the scalability of the system by the ratio of w and s , which we note as $\tau = w/s$. For a well scalable system, despite the increasing number of nodes and local schedulers, τ should remain $\ll 1$, so that the system can adapt well with the workload change. It would be interesting to look at how the problematic architecture of a system requires additional information and feedbacks to achieve the same schedul-

ing goal, thus causes s to increase, and ultimately limits how large (the scheduling component of) the system can scale to. Such study would provide valuable insights on the trade-offs between the scalability of a system, the performance guarantee it provides, and the engineering effort required to achieve such scalability and performance.

6.3.2 Completeness of the Maat Principle: A Formal Analysis

In this dissertation, we first introduced the CLI ideal scheduling conditions: Completeness, Local enforceability, and Independent scheduling points. We then identified five common problems that cause CLI violations and thus lead to scheduling failures.

Some questions naturally follow: are the five problems the only ones that can prevent effective scheduling in a system? If a system is free of the five problems, can any (reasonable) scheduling policy be realized on it without difficulties? In other words, is the Maat principle *complete*?

One can answer the completeness questions empirically by examining multiple systems and experimenting with various scheduling policies, as we did in this dissertation. Fully answering these questions, however, requires a more formal treatment of the scheduling problem.

The CLI ideal scheduling conditions can serve as a good starting point; however, it is still drawn from our experiences and intuition, instead of derived from the first principle of scheduling. A more precise definition of "inadequate scheduling support" is needed to separate the difficulties inherent to a scheduling goal (e.g., achieve isolation and high utilization simultaneously), from the scheduling difficulties introduced by the system structure (e.g., schedulers being placed at the wrong spot in the system). The queuing network formulation is useful in this context: the system can be modeled as a queuing network, and different scheduling goals as different utility functions. Given the definition, a stronger link from

inadequate scheduling and the CLI conditions can be established. The completeness of the five Maat problems can then be reasoned by examining exhaustively how thread behaviors and interactions can violate the CLI conditions.

Given the thread architecture of a system, ideally one should be able to use the above formulation to derive the list of scheduling policies that can be implemented in this system, how precise the scheduling control would be for these policies, and why other scheduling policies are not possible to realize in this system.

6.3.3 Build a Natively Schedulable Storage System

As we discussed earlier (§6.2), schedulability should be made a first-class citizen in system design. In this dissertation we mostly focus on retrofitting scheduling controls into existing systems, which did not consider schedulability in their initial design processes. Many scheduling problems arise as a result; we discuss possible solutions to eliminate or mitigate these problems in this dissertation.

Another possible direction is to construct a new system from scratch that adheres to the Maat principle. Such a natively schedulable system (as opposed to the systems that require restructuring or extra feedback mechanisms to enable scheduling) allows the maximum flexibility in realizing various scheduling policies. Since no control plane involvement is required to mitigate structural scheduling problems, the system would also scale much better than traditional systems, which only add scheduling as an afterthought and introduce significant control plane overheads to do so.

6.4 Closing Words

Resource sharing and scheduling has always been one of the key problems in computer sciences. With cloud computing on the rise, it is becoming even more important. Storage systems, however, present some unique challenges in terms of scheduling the shared resource accesses.

People have developed numerous schedulers to achieve various performance goals; these schedulers, however, are only effective when placed at the right place and given the right support in the system. In this dissertation we looked at multiple systems, including the local and distributed ones, where the traditional approaches of scheduling fail to provide such support and thus lead to scheduling failures. We hope that in the future the community could put more thoughts on the scheduling support a system provides (which we term the *schedulability* of the system) and design the next generation of storage systems with strong performance guarantees.

Bibliography

- [1] BPF Compiler Collection (BCC). <https://github.com/iovisor/bcc>.
- [2] CFQ (Complete Fairness Queueing). <https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt>.
- [3] Database/kernel community topic at collaboration summit 2014. <http://www.postgresql.org/message-id/20140310101537.GC10663@suse.de>.
- [4] Deadline IO scheduler tunables. <https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt>.
- [5] Documentation for pgbench. <http://http://www.postgresql.org/docs/9.4/static/pgbench.html>.
- [6] Documentation for /proc/sys/vm/*. <https://www.kernel.org/doc/Documentation/sysctl/vm.txt>.
- [7] Ext4 Filesystem. <https://www.kernel.org/doc/Documentation/filesystems/ext4.txt>.
- [8] Inside the Windows Vista Kernel: Part 1. <http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx>.

- [9] ionice(1) - Linux man page. <http://linux.die.net/man/1/ionice>.
- [10] Notes on the Generic Block Layer Rewrite in Linux 2.5. <https://www.kernel.org/doc/Documentation/block/biodoc.txt>.
- [11] postgresql-hackers maillist communication. http://www.postgresql.org/message-id/CA+Tgmobv6gm6SzHx8e2w-0180+jHbCNYbAot9KyzG_3DxRYxaw@mail.gmail.com.
- [12] Postgresql 9.2.9 documentation. <http://www.postgresql.org/docs/9.2/static/wal-configuration.html>.
- [13] HBase Issue Tracking Page. <https://issues.apache.org/jira/browse/HBASE-8884>, July 2013.
- [14] Cassandra Issues: Move away from SEDA to TPC. <https://issues.apache.org/jira/browse/CASSANDRA-10989>, Sep 2016.
- [15] Cassandra Issues: User based request scheduler. <https://issues.apache.org/jira/browse/CASSANDRA-8032>, January 2016.
- [16] HBase Issue Tracking Page. <https://issues.apache.org/jira/browse/HBASE-8836>, July 2016.
- [17] MongoDB Issue Tracking Page. <https://jira.mongodb.org/browse/SERVER-24661>, July 2016.
- [18] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina Popovici, Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Suman Banerjee. Avoiding file system micromanagement with range writes. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 161–176. USENIX Association, 2008.

- [19] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O’Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, pages 233–248, 2014.
- [20] Andrea C Arpaci-Dusseau and Remzi H Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 43–56. ACM, 2001.
- [21] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.
- [22] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999.
- [23] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The data-center as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.
- [24] Steve Best. JFS Overview. <http://jfs.sourceforge.net/project/pub/jfs.pdf>, 2000.
- [25] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.
- [26] P. Bosch and S.J. Mullender. Real-time disk scheduling in a mixed-media file system. In *Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE*, pages 23–32, 2000.
- [27] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

- [28] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Multimedia Computing and Systems, 1999. IEEE International Conference on*, volume 2, pages 400–405 vol.2, Jul 1999.
- [29] Florian Buchholz. The structure of the Reiser file system. <http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php>, January 2006.
- [30] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.
- [31] David D Chambliss, Guillermo A Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P Lee. Performance virtualization for large-scale storage systems. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 109–118. IEEE, 2003.
- [32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
- [33] Dah-Ming Chiu and Raj Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.
- [34] Kristina Chodorow. *MongoDB: the Definitive Guide*. " O'Reilly Media, Inc.", 2013.

- [35] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
- [36] Silviu S. Craciunas, Christoph M. Kirsch, and Harald Röck. The TAP Project: Traffic Shaping System Calls. <http://tap.cs.uni-salzburg.at/downloads.html>.
- [37] Silviu S. Craciunas, Christoph M. Kirsch, and Harald Röck. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev.*, 42(5):44–54, July 2008.
- [38] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in multics. *Communications of the ACM*, 11(5):306–312, 1968.
- [39] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.
- [40] Zoran Dimitrijevic, Raju Rangaswami, and Edward Y Chang. Design and implementation of semi-preemptible io. In *FAST*, 2003.
- [41] Andrew E Dinn. Flexible, dynamic injection of structured advice using byteman. In *Proceedings of the tenth international conference on Aspect-oriented software development companion*, pages 41–50. Acm, 2011.
- [42] Michael Warrilow Ed Anderson. Market insight: Cloud shift – the transition of it spending from traditional systems to cloud.
- [43] Bryan Fink. Distributed computation on dynamo-style distributed storage: riak pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 43–50. ACM, 2012.

- [44] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 307–320. ACM, 2007.
- [45] Gregory R. Ganger, Marshall Kirk McKusick, Craig A.N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.
- [46] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. " O'Reilly Media, Inc.", 2011.
- [47] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, volume 11, pages 24–24, 2011.
- [48] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *USENIX Annual Technical Conference*, pages 459–471, 2015.
- [49] Pawan Goyal, Xingang Guo, and Harrick M Vin. A hierarchical cpu scheduler for multimedia operating systems. In *OSDI*, volume 96, pages 107–121, 1996.
- [50] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '96*, pages 157–168, New York, NY, USA, 1996. ACM.
- [51] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clus-

- ters. In *Proceedings of OSDITM16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 65, 2016.
- [52] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of the USENIX OSDI*, 2016.
- [53] Andrew Grimshaw, Adam Ferrari, Frederick Knabe, and Marty Humphrey. Wide area computing: resource sharing on a large scale. *Computer*, 32(5):29–37, 1999.
- [54] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K Iyer, Zhen-Yu Yang, et al. Characterization of linux kernel behavior under errors. In *DSN*, volume 3, pages 22–25, 2003.
- [55] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, volume 9, pages 85–98, 2009.
- [56] Ajay Gulati, Arif Merchant, and Peter J. Varman. 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, SIGMETRICS '07*, pages 13–24, New York, NY, USA, 2007. ACM.
- [57] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 437–450. USENIX Association, 2010.
- [58] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*, pages 342–362. Springer, 2006.

- [59] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, SOSP '87*, pages 155–162, New York, NY, USA, 1987. ACM.
- [60] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
- [61] D. Richard Hipp and D. Kennedy. SQLite, 2007.
- [62] Micha Hofri. Disk scheduling: Fcfs vs. sstf revisited. *Communications of the ACM*, 23(11):645–653, 1980.
- [63] L. Huang and T. Chiueh. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, March 2000.
- [64] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.
- [65] David M Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Hewlett-Packard Laboratories Palo Alto, CA, 1991.
- [66] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 191–202, Santa Clara, CA, 2015. USENIX Association.
- [67] Maulana Karenga. *Maat, The Moral Ideal in Ancient Egypt: A Study in Classical African Ethics*. Routledge, 2003.

- [68] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Disk Schedulers for Solid State Drivers. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 295–304, New York, NY, USA, 2009. ACM.
- [69] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the i/o path: A holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 345–358, Santa Clara, CA, 2017. USENIX Association.
- [70] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.
- [71] Rusty Klophaus. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.
- [72] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 1–14. ACM, 2015.
- [73] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [74] Jim Larson. Erlang for concurrent programming. *Communications of the ACM*, 52(3):48–56, 2009.

- [75] Jacob R Lorch and Alan Jay Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 143–154. ACM, 1996.
- [76] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
- [77] Christopher R Lumb, Arif Merchant, and Guillermo A Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*, volume 3, pages 131–144, 2003.
- [78] Christopher R Lumb, Jiri Schindler, and Gregory R Ganger. Free-block scheduling outside of disk firmware.
- [79] Christopher R Lumb, Jiri Schindler, Gregory R Ganger, David F Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 7. USENIX Association, 2000.
- [80] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *NSDI*, pages 589–603, 2015.
- [81] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 144–159. ACM, 2016.

- [82] Chris Mason. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.
- [83] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.
- [84] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August, 2:2009*, 2008.
- [85] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 93, 1993.
- [86] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.
- [87] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
- [88] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.
- [89] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST*, page 13, 2012.
- [90] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *OSDI*, pages 433–448, 2014.

- [91] Florentina I Popovici, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *USENIX Annual Technical Conference, General Track*, pages 297–310, 2003.
- [92] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 13–25. ACM, 2008.
- [93] Alma Riska, James Larkby-Lahet, and Erik Riedel. Evaluating block-level optimization through the io path. In *USENIX Annual Technical Conference*, pages 247–260, 2007.
- [94] Luigi Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.
- [95] Luigi Rizzo and Fabio Checconi. GEOM_SCHED: A Framework for Disk Scheduling within GEOM. http://info.iet.unipi.it/~luigi/papers/20090508-geom_sched-slides.pdf.
- [96] Lawrence G Roberts and Barry D Wessler. Computer network development to achieve resource sharing. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, pages 543–549. ACM, 1970.
- [97] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.
- [98] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [99] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. 27(3):17–28, March 1994.

- [100] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference*, pages 313–323. Washington, DC, 1990.
- [101] Prashant J Shenoy and Harrick M Vin. Cello: A disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 44–55. ACM, 1998.
- [102] David Shue and Michael J Freedman. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems*, page 17. ACM, 2014.
- [103] David Shue, Michael J Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, volume 12, pages 349–362, 2012.
- [104] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.
- [105] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.
- [106] Lalith Suresh Puthalath. On predictable performance for distributed systems. 2016.
- [107] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

- [108] Andrew S. Tanenbaum. *Computer Networks (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [109] Eno Thereska, Hitesh Ballani, Greg O’Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.
- [110] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor. Technical report, Tech. Rep., May, 2011.
- [111] Theodore Ts’o. <http://e2fsprogs.sourceforge.net>, June 2001.
- [112] Rodney Van Meter and Minxi Gao. Latency management in storage systems. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 8. USENIX Association, 2000.
- [113] Kenton Varda. Protocol buffers: Google’s data interchange format. *Google Open Source Blog*, Available at least as early as Jul, 2008.
- [114] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, volume 7, pages 5–5, 2007.
- [115] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.
- [116] Carl A Waldspurger and William E Weihl. *Stride Scheduling: Deterministic Proportional Share Resource Management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

- [117] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 14. ACM, 2012.
- [118] Hui Wang and Peter J Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST*, pages 229–242, 2014.
- [119] Aaron Weiss. Computing in the clouds. *networker*, 11(4):16–25, 2007.
- [120] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.
- [121] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [122] Bruce L Worthington, Gregory R Ganger, and Yale N Patt. Scheduling algorithms for modern disk drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 22, pages 241–251. ACM, 1994.
- [123] Yiqi Xu and Ming Zhao. IBIS: Interposed Big-Data I/O Scheduler. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 111–122. ACM, 2016.
- [124] Yuehai Xu and Song Jiang. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In *In 9th USENIX Conference on File and Storage Technologies. USENIX*. Citeseer, 2011.

- [125] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 131–146. USENIX Association, 2006.
- [126] Ting Yang, Tongping Liu, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Redline: First class support for interactivity in commodity operating systems. In *OSDI*, volume 8, pages 73–86, 2008.
- [127] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y Wang, Kai Li, Arvind Krishnamurthy, and Thomas E Anderson. Trading capacity for performance in a disk array. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 17. USENIX Association, 2000.
- [128] Jiaan Zeng and Beth Plale. Workload-aware resource reservation for multi-tenant nosql. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 32–41. IEEE, 2015.
- [129] Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y Wang. Configuring and scheduling an eager-writing disk array for a transaction processing workload. In *Proceedings of the 1st USENIX conference on File and storage technologies*, pages 21–21. USENIX Association, 2002.
- [130] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.