IMPROVING AND TUNING THE PERFORMANCE OF SERVER SYSTEMS

BY Cheng Li

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Thu D. Nguyen and Ricardo Bianchini

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2015

ABSTRACT OF THE DISSERTATION

Improving and Tuning the Performance of Server Systems

by Cheng Li

Dissertation Directors: Thu D. Nguyen and Ricardo Bianchini

Modern server systems incorporate complex hardware and software technologies, such as solid-state drives and software virtualization. Maximizing the performance of these complex systems involves many challenges. For example, their performance can often be strongly affected by multiple configuration parameters. At the same time, service providers using these servers often have complex performance objectives, such as achieving multiple performance targets at the same time.

In this dissertation, we address three such challenges: improving space efficiency in solid-state drives used as caches (and consequently performance) for storage arrays, reducing performance variability in virtualized systems, and engineering performance to meet multiple performance targets. We built three systems to tackle these challenges concretely. The first system, called Nitro, uses deduplication and local compression to increase the effective (solid-state) cache size of network-attached storage systems, improving performance and space efficiency while minimizing total cost. The second system, called VirtualFence, leverages solid-state drives and non-work-conserving scheduling to provide consistent I/O performance in virtualized multi-tenant systems. The third system, called OpTune, mitigates the complexity of

tuning performance to meet multiple performance targets in multi-tier systems using optimization. Our extensive experimental evaluations show that our systems can consistently improve performance and/or achieve the desired performance objectives. These positive results suggest that the principles and techniques embodied in our systems are strong steps toward effectively managing the performance of modern server systems.

Acknowledgements

First and foremost, I would like to thank my research advisor, Ricardo Bianchini, whose guidance, encouragement, and support made the dissertation possible. Not only did he worked closely with me on all research topics, but also his passion towards work and life greatly influenced me. I learned a lot from him, including problem solving, creative thinking, dedication and tenacity. These skills and merits help build solid foundations for my future career.

I would also like to thank my research co-advisor, Thu D. Nguyen, for the consistent help during the entire course of my PhD. Thu helps me to make the transition from an inexperienced graduate student to a computer science researcher. I enjoyed every discussion with him on technical details, research ideas and high-level technology trend.

I would like to thank my thesis committee members Abhishek Bhattacharjee, for his advice on how to present and write papers, and Dr. Hui Zhang for valuable feedback on the dissertation. I also would like to thank to my internship mentors and collaborators. I like to thank immensely to Philip Shilane, Fred Douglis, Grant Wallace and Stephen Smaldone at EMC Princeton office. I like to thank Hyong Shim, Windsor Hsu, Darren Sawyer, and Stephen Manley at EMC; Neil Blakery-Milner, Brian Pane, Federico Larumbe, Jonathan Frank, Peter Ruibal and Doug Beaver at Facebook. The internships helped me understand real world problems and helped me identify interesting research topics.

It is a pleasure to work in DARK and PANIC lab, where hard work and fun coexist. It is a great pleasure to have worked with my labmates, Rekha Bachwani, Josep Lluis Berral, Guilherme Cox, Qingyuan Deng, Inigo Goiri, Md E. Haque, Bill Katsaks, Kien Le, Ioannis Manousakis, Luiz Ramos, and Wei Zheng. I enjoyed open discussion, brainstorming, and

friendship over the course of these years. Most importantly, you made the challenging, sometimes frustrating process much easier to go through.

Finally, I would like to deeply thank my parents and relatives. Their support helped me go through the ups and downs in the course of my PhD years.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables x

List of Figures xi

1. Introduction 1

1.1. Motivation 1

1.2. Research Overview 5

1.2.1. Nitro 5

1.2.2. VirtualFence 5

1.2.3. OpTune 6

1.3. Contributions 6

1.4. Overview of the Dissertation 7

2. Background and Related Work 8

2.1. Optimizing Flash Performance and Capacity 8

2.2. I/O Predictability 9

2.3. Performance Engineering in Server Systems 11

3. A Capacity-Optimized SSD Cache for Primary Storage 13

3.1. Introduction 13

3.2. Background and Discussion 15
3.3. Nitro Architecture and Design 17
3.3.1. Nitro Components 19
3.3.2. Nitro Functionality 21
3.4. Nitro Implementation 24
3.5. Experimental Methodology 25
3.5.1. Metrics 26
3.5.2. Experimental Traces 26
3.5.3. Parameter Space 28
3.5.4. Experimental Platform 28
3.6. Evaluation 29
3.6.1. Simulation Results 29
3.6.2. Prototype System Results 33
3.6.3. Nitro Advantages 38
3.7. Summary 40

4. I/O Predictability in Virtualized Multi-tenant Systems 41
4.1. Introduction 41
4.2. Motivation 43
4.3. Methodology 45
4.3.1. Performance Deviation Metrics 46
4.3.2. Virtualized Systems 46
4.3.3. Workloads 47
4.3.4. Experimental Platform 48
4.4. VMM-Driven Performance Deviation 49
4.5. VirtualFence 51

4.5.1. Prototype . . . 52
4.5.2. Space Partitioning . . . 53
4.5.3. Time Partitioning . . . 54
4.6. Evaluation . . . 56
4.6.1. Performance Deviation . . . 57
4.6.2. Performance Deviation at Low Load . . . 60
4.6.3. Performance vs. Predictability . . . 62
4.6.3.1. Impact of Number of Slots . . . 63
4.6.3.2. Impact of Slot Length . . . 63
4.7. Discussion . . . 64
4.8. Summary . . . 66

5. Multi-Point Performance Engineering in Server Systems . . . 67
5.1. Introduction . . . 67
5.2. OpTune Methodology . . . 70
5.2.1. Overview . . . 70
5.2.2. Performance Composition . . . 73
5.2.3. Performance Decomposition . . . 75
5.2.4. Implementing OpTune . . . 76
5.3. OpTune Systems . . . 78
5.3.1. Web Server . . . 78
5.3.2. Filesystem Emulator . . . 80
5.3.3. MapReduce System . . . 82
5.4. Evaluation . . . 83
5.4.1. Experimental Setup . . . 83
5.4.2. Impact of Configuration Parameters . . . 85

5.4.3. Performance Tuning 88
5.4.4. Sensitivity Analysis 91
5.5. Summary 91

6. Conclusion and Future Work 93

Appendix A. Additional Evaluation for OpTune 96
A.1. Filesystem Emulator with OpTune 96
A.1.1. Independent Tests 96
A.1.2. Single-point Performance Optimization 97
A.1.3. Multi-point performance optimization 98
A.2. MapReduce with OpTune 99
A.2.1. Single-point Performance Optimization 99
A.2.2. Multi-point performance optimization 100
A.3. OpTune Solver Overhead Analysis 100
A.3.1. Solving Time 101
A.3.2. Convergence Speed 101
A.3.3. Extrapolation of Solving Time 102
References 104

List of Tables

3.1. Parameters for Nitro with default values in bold. 28

3.2. Performance evaluation of Nitro and its variants. 34

4.1. VirtualFence variants 57

5.1. Detailed results for web server. 89

A.1. Detailed results for filesystem emulator. 98

A.2. Detailed results for MapReduce system. 100

A.3. Running time of different tuning tasks. 101

List of Figures

3.1. Potential deduplication in cache. 15

3.2. SSD cache and disk storage. 18

3.3. Nitro indices 20

3.4. Read-hit ratio of WEU-based vs. Extent-based. 30

3.5. Fingerprint index ratio impact. 31

3.6. Number of SSD erasures for modified and unmodified SSD variants. . . 32

3.7. Sensitivity analysis of (D, NC) and (D, C). 36

3.8. Overheads of Nitro prototypes. 37

3.9. Response time diverges for random I/Os. 38

3.10. Nitro improves snapshot restore performance. 40

4.1. Performance deviation 4-VM heterogeneous workloads. 49

4.2. VirtualFence architecture. 52

4.3. Driver with non-work-conserving time partitioning. 54

4.4. Deviation when running the 4-VM heterogeneous workloads 57

4.5. Deviation of VirtualFence. 60

4.6. Deviation when running 4-VM homogeneous, low-rate I/O workloads. 61

4.7. Number of slots and response time trade-off. 62

5.1. Impact of a cache configuration parameter on the response time of a Web server. 68

5.2. The service graph of a Web server. 71

5.3. An illustration of periodic activities in OpTune. 72

5.4. Sequential (a), conditional (b), parallel (c), and loop patterns. 73

5.5. File server composition graph. 81

5.6. MapReduce composition graph. 82

5.7. The SWIM trace and utilization 85

5.8. Impact of $Size(f)$ on the Web server's response time. 86

5.9. Impact of `idle_time` on the Web server's response time. 87

5.10. Illustration of independence 88

5.11. Validation of prediction. 89

5.12. Web server's response time for different performance objectives. 89

5.13. Multi-target results. 90

5.14. Impact of I/O parameter. 92

A.1. Illustration of independence 97

A.2. Filesystem emulator's response time for different performance objectives. . . . 98

A.3. MapReduce's response time for different performance objectives. 99

A.4. Solution quality over time using brute-force and simulated annealing. 102

A.5. Time complexity for OpTune solver. 103

Chapter 1

Introduction

Despite having received a significant amount of attention from researchers and practitioners, server systems still exhibit some important performance challenges, such as inefficiencies in the use of solid-state drives (SSDs) as data caches, performance interference and variability in virtualized systems, and difficulties in configuring systems to meet multiple performance targets. In this dissertation, we address these three challenges by designing, implementing, and evaluating performance improvement and tuning techniques for several real systems. Next, we overview the motivation for our efforts, and the techniques and systems we built. We conclude the section with a summary of our contributions, and an outline for the rest of the dissertation.

1.1 Motivation

Server systems are the basic but crucial components in datacenters. Internet service giants, such as Google, Facebook, and Amazon, manage hundreds of thousands of server machines in their datacenters. We observe three trends from this big picture: (1) As advanced hardware technologies (*i.e.*, SSDs, phase-change memory) become available, more and more server systems will leverage these advanced technologies to improve overall performance; (2) cloud service providers, such as Google App Engine, Amazon EC2, and Microsoft Azure, allow small companies to rent servers to help traditional IT department to transition their computing systems to the cloud; (3) As server systems gets complicated, performance tuning for these systems needs to consider multiple factors or multiple dimensions, which are based on users new requirements. Next, we summarize the motivation of the thesis and new challenges from

the architectural and performance requirement perspectives.

Inefficiency in the use of SSD cache space. The emergence of flash-based SSD reshapes the storage landscape because SSD as a persistent storage that has much higher performance than hard disk drives (HDD). This can help service provider significantly increase the performance of their servers. However, there are inherent tradeoffs between the performance and cost per byte of storage. For example, a 2.4 TB flash fusion I/O card that can service ≥100K IOPS nearly costs ~$27K [2], while 2TB HDD that barely services ≤100 IOPS only costs ~$100. Fundamentally though, the goals of high performance and cost-efficient storage are in conflict. Solid state drives (SSDs) can support high IOPS with low latency, but their cost will be higher than hard disk drives (HDDs) for the foreseeable future [85]. In contrast, HDDs have high capacity at relatively low cost, but IOPS and latency are limited by the mechanical movements of the drive. Although a full-flash array can increase performance significantly, system administrators may find a HDD/SSD hybrid storage array more cost-efficient, where SSD services as a cache in front of HDD arrays.

Previous work has explored SSDs as a cache in front of HDDs to address performance concerns [7, 37, 113], and the SSD interface has been modified for caching purposes [93], but the cost of SSDs continues to be a large fraction of total storage cost. Thus, further optimizing SSDs caching performance along both the space and cost dimensions remains an open research problem.

I/O unpredictability in virtualized multi-tenant systems. Recent trend shows that IT consumers prefer to rent servers and resources rather than spending capital on these dedicated infrastructure and depreciate over time. The notion of cloud computing is based on sharing of resources to achieve economic of scale. IT infrastructure is outsourced to cloud service providers so that workloads from different IT consumers are shared and collocated. Traditional IT infrastructure model is shrinking because cloud computing enables IT to more rapidly adjust resources to meet fluctuating and unpredictable business demand.

With the advent of cloud computing, virtualization has become the primary strategy to consolidate diverse workloads (encapsulated in virtual machines or VMs) to ensure high utilization of physical machines (PMs). The workload in the cloud ranges from computation intensive AI tasks [28, 34, 112, 111] to HPC jobs [77, 50]. Among its various benefits, virtualization increases fault isolation and simplifies workload migration [13, 14]. Many IaaS cloud providers, such as Amazon EC2 and Rackspace, use virtualization and consolidation in offering their services.

Unfortunately, our quantification of storage I/O performance across a range of workloads, virtual machine monitor (VMM) architectures, approaches to storage virtualization, and storage devices shows widespread performance *unpredictability* in the face of consolidation. In fact, VM performance is essentially unpredictable, since the number of co-located VMs and their workloads may change each time the VM runs, or even during a single run. For example, researchers have shown a single run of a fixed-load VM on Amazon EC2 to exhibit wild performance swings due to consolidation [87]. VM performance may vary significantly in the face of consolidation. Interestingly, the use of solid-state drives (SSDs) can exacerbate the problem because the SSD write performance gets much worse when consolidated with multiple VMs. Since many users may desire consistent performance, we argue that IaaS cloud providers should offer a new class of predictable-performance service *in addition to* (and using different resources from) their existing (predictability-oblivious) services.

Along these lines, our research seeks to create virtualized systems that exhibit *performance predictability* for this new class of service. This property implies that the average throughput and response time experienced by each VM should be unaffected by any other VM executing on the same PM or the overall utilization of the PM. In fact, performance should be the same whether the VM runs in isolation or is co-located with any number (up to a pre-defined limit) of other VMs.

Importantly, note that our notion of performance predictability differs from *performance*

isolation [14, 45, 46, 59, 62, 86, 106]. The goal of isolation is to ensure that each co-located VM achieves at least a minimum desired level of performance. This is one of the goals of predictability. However, in performance isolation it is typically acceptable to dedicate more resources than this minimum, if those resources are available. In contrast, dedicating any available resources beyond a fixed amount will likely ruin predictability. One may see performance isolation as a less strict form of performance predictability. The performance of each VM should always be the same, regardless of how many other VMs are running on the same server or how active those VMs are.

Challenges in simultaneously achieving multi-point performance objectives. Modern server systems encompass multiple components and/or layers containing configuration parameters that can affect performance. Examples include parameters controlling the amount of parallelism (*e.g.*, number of threads), the size and replacement policy used for memory caches, and the scheduling policies for processing workloads. As the complexity of server systems continues to increase, managing the interplay between these configuration parameters to precisely tune performance becomes a challenging task.

This challenge is exacerbated by the need of many service providers to meet multiple performance objectives. For example, reducing the tail latencies of on-line services has received much attention (e.g., [31, 47, 116]). However, techniques and configuration parameter values for reducing tail latencies can often negatively impact performance at other percentiles in the performance cumulative distribution function (CDF).

Given the challenge of tuning system performance to meet a single performance objective (*e.g.*, minimize median response time), it is not surprising that tuning for multiple performance targets is a challenging, error-prone, and time-consuming exercise for server system administrators.

1.2 Research Overview

With the above motivations in mind, the goal of this dissertation is to build software techniques and systems that mitigate the inefficiencies and performance management challenges we identified. Specifically, we design and prototype three novel systems: Nitro, VirtualFence, and OpTune. The next few subsections overview the systems and their main evaluation results.

1.2.1 Nitro

Nitro is an SSD cache that leverages data reduction techniques to further reduce the capital cost of and to increase effective capacity. Data reduction techniques adopted in Nitro include deduplication and local compression.

We designed and built Nitro, and explored extensively with evaluation: (1) an SSD cache design with adjustable deduplication, compression, and large replacement units, (2) an evaluation of the trade-offs between data reduction, RAM requirements, SSD writes (reduced up to 53%, which improves lifespan), and storage performance, and (3) acceleration of two prototype storage systems with an increase in IOPS (up to 120%) and reduction of read response time (up to 55%) compared to an SSD cache without Nitro. Additional benefits of Nitro include improved random read performance, faster snapshot restore, and reduced writes to SSDs.

1.2.2 VirtualFence

VirtualFence is a storage system that provides predictable performance in virtualized multi-tenant scenario. In this work, we argue that IaaS cloud providers should provide a class of predictable-performance service in addition to their existing (predictability-oblivious) services since many users may desire consistent performance.

VirtualFence reduces I/O unpredictability by leveraging three main techniques: (1) non-work-conserving time-division I/O scheduling, (2) a small SSD cache in front of a much larger hard disk drive (HDD), and (3) space-partitioning of both the SSD cache and the HDD.

Our evaluation of VirtualFence demonstrates that improves I/O predictability in virtualized systems significantly. More fundamentally, VirtualFence illustrates the tradeoff between predictability and performance. We conclude that current VMMs are far from providing storage I/O performance predictability. Systems like VirtualFence can remedy this problem, while allowing the cloud provider to select an appropriate compromise between performance and predictability.

1.2.3 OpTune

OpTune is a framework designed to help system administrators tune these parameters. OpTune asks administrators to specify objectives to shape the performance CDFs of systems; *e.g.*, minimize the median response time while keeping the 99^{th} percentile below a target value. In fact, administrators can specify entire target CDFs. OpTune then uses a graphical representation of the system, performance instrumentation and profiling, and manipulations of the profiled CDFs of components to configure the system.

We demonstrate the broad utility of OpTune by integrating it into three different, widely-used systems: a Web server, a filesystem emulator, and a MapReduce server cluster. Evaluation results demonstrate that OpTune successfully helps administrators to quickly identify configuration parameter values to best achieve the desired performance behaviors.

1.3 Contributions

In summary, our contributions in this dissertation are:

- We propose Nitro, an SSD cache that utilizes deduplication, compression, and large replacement units to accelerate primary I/O.
- We investigate the trade-offs between deduplication, compression, RAM requirements, performance, and SSD lifespan. We experiment with storage system prototypes to validate Nitro's performance improvements.

- We study the impact of VMM architecture, approach to storage virtualization, and storage device on I/O predictability.
- We propose VirtualFence, a system that combines SSDs working as HDD caches, space and non-work-conserving time partitioning. We quantify the impact of each feature of VirtualFence on I/O predictability. Using VirtualFence, we investigate the fundamental tradeoff between predictability and performance.
- We propose and build the OpTune framework for guiding administrators when configuring server systems to meet a set of performance objectives.
- We implement OpTune in three diverse server systems to demonstrate its wide applicability.
- We present results from a large set of case studies to show how OpTune can ease the task of performance tuning, particularly when this process involves tradeoffs between multiple points on the performance CDF (*e.g.*, average and/or median vs. tail latencies).

1.4 Overview of the Dissertation

The remainder of the dissertation processes as follows. Chapter 2 discusses related works. Chapter 3 describes the design, implementation, and evaluation of Nitro. Chapter 4 details the I/O predictability problem in virtualized multi-tenant systems and describes VirtualFence. Chapter 5 presents the details and evaluation OpTune. Finally, Chapter 6 concludes the dissertation and discusses future work.

Chapter 2

Background and Related Work

In this chapter, we first describe prior efforts on SSD caches research to provide background for the remainder of the thesis. We then discuss efforts and related work in I/O predictability research and finally summarize related work in admin-guided full-range performance engineering.

2.1 Optimizing Flash Performance and Capacity

SSD as storage or cache. Many studies have focused on incorporating SSDs into the existing hierarchy of a storage system [11, 37, 58, 113]. In particular, several works propose using flash as a cache to improve performance. For example, Intel Turbo Memory [78] adds a nonvolatile disk cache to the hierarchy of a storage system to enable fast start-up. Kgil et al. [57] splits a flash cache into separate read and write regions and uses a programmable flash memory controller to improve both performance and reliability. However, none of these systems combine deduplication and compression techniques to increase the effective capacity of an SSD cache.

Several recent papers aim to maximize the performance potential of flash devices by incorporating new strategies into the established storage I/O stack. For example, SDF [88] provides a hardware/software co-designed storage to exploit flash performance potentials. FlashTier [93] specifically redesigned SSD functionality to support caching instead of storage and introduced the idea of silent eviction. As part of Nitro, we explored possible interface changes to the SSD including aligned WEU writes and TRIM support, and we measured the impact on SSD lifespan.

Deduplication and compression in SSD. Deduplication has been applied to several primary storage systems. iDedup [99] selectively deduplicates primary workloads inline to strike a balance between performance and capacity savings. ChunkStash [32] designed a fingerprint index in flash, though the actual data resides on disk. Dedupv1 [81] improves inline deduplication by leveraging the high random read performance of SSDs. Unlike these systems, Nitro performs deduplication and compression within an SSD cache, which can enhance the performance of many primary storage systems.

Deduplication has also been studied for SSD storage. For example, CAFTL [24] is designed to achieve best-effort deduplication using an SSD FTL. Kim et al. [60] examined using the on-chip memory of an SSD to increase the deduplication ratio. Unlike these systems, Nitro performs deduplication at the logical level of file caching with off-the-shelf SSDs. Feng and Schindler [38] found that VDI and long-term CIFS workloads can be deduplicated with a small SSD cache. Nitro leverages this insight by allowing our partial fingerprint index to point to a subset of cached entries. Another distinction is that since previous deduplicated SSD projects worked with fixed-size (non-compressed) blocks, they did not have to maintain multiple references to variable-sized data. Nitro packs compressed extents into WEUs to accelerate writes and reduce fragmentation. SAR [76] studied selective caching schemes for restoring from deduplicated storage. Our technique uses recency instead of frequency for caching.

2.2 I/O Predictability

Disk Drives and I/O Interference. Many recent studies have established that I/O interference prevents VMs from achieving predictable performance [14, 45, 46, 59, 62, 106]. This is primarily due to the mechanical nature of HDDs. Disks make I/O performance highly dependent on the locality of accesses across VMs, variability in I/O sizes, request priorities, and access burstiness [45].

Many previous efforts to address this problem have focused primarily on resource scheduling techniques, seeking to provide proportional allocation of I/O resources with strong isolation [48, 53, 104, 109]. Argon [104] in particular shares common techniques with VirtualFence, such as space partitioning of caches (memory caches in the case of Argon) and time partitioning of I/O access time. However, our focus on predictability instead of isolation leads to fundamental differences, including non-work-conserving allocation policies and static configuration parameters.

Other works seek to provide proportional allocation while supporting latency-sensitive applications [36, 91, 109, 117]. mClock [45] goes further by providing proportional allocation, subject to minimum reservations and maximum limits. In principle, mClock can support predictability when the maximum limit is set equal to the reservation. However, the authors did not consider this scenario. Moreover, mClock does not consider the properties of storage devices and how they impact predictability. VirtualFence does and, hence, combines SSDs and HDDs. For this reason, the two systems cannot be fairly compared quantitatively. Alternate approaches using sophisticated machine learning techniques to meet end-user QoS targets have also been studied [118].

Besides the differences described above, our work identifies predictability, a stricter form of isolation, as desirable, and is the first to study hybrid SSD/HDD systems in this context. We also quantify the lack of predictability across a range of VMM architectures and configurations.

Hybrid SSDs and HDDs. Recent advances in Flash memory SSDs have led to studies exploiting their high-speed random reads, low power consumption, feature size, and shock resistance [7, 15, 35, 56, 67, 110]. Most research on SSD organization has focused on either using SSDs as HDD replacements [15, 54], or using SSDs as a caching layer in the storage hierarchy [56, 67, 85, 92, 113]. The primary focus of these works is on the performance benefits of SSDs. The energy implications of SSDs have also been studied [21, 103].

In comparison to these efforts, our work is the first to quantify the impact of SSDs on

VM performance predictability. In fact, we present the first characterization of predictability on SSDs across a spectrum of VMM types. Interestingly, our results show that SSDs do *not* guarantee predictability: in write-intensive workloads, their predictability is actually *worse* than that of HDDs. Moreover, SSDs exhibit cost-per-byte that still cannot compete with those of HDDs. VirtualFence combines the advantages of both storage media to promote predictability.

2.3 Performance Engineering in Server Systems

Full-range performance tuning has recently garnered interest. For example, recent work [119] uses a similar approach to OpTune—*i.e.*, graphical representation of system together with composition of components' performance CDFs—to predict the overall performance of compound Web services. Unlike this study however, we seek to tune configuration parameters to actually achieve performance goals rather than only estimating system performance. We also implement and evaluate OpTune in three real systems.

Several additional projects have studied the problem of performance variability in different ways. For example, the real-time systems community has considered mechanisms to ensure that groups of tasks meet their deadline targets [18, 30], while the high performance computing community has studied mechanisms to remove performance variability or jitter [96]. OpTune is more flexible in that it helps administrators to shape the entire performance CDFs. Thus, it can be used to find appropriate configurations for different desired tradeoffs between performance (e.g., average response time) and performance variability (e.g., tail latencies).

Our work is also partly inspired by prior work on quality of service and resource allocation fairness studies [23, 65, 12, 41, 45], which attempt to guarantee a minimum level of performance in server and networked systems. While these approaches do use optimization techniques to suggest system configuration settings, they incentivize users to share resources, leaving other hardware idle. As pointed out in [90, 95] however, this may increase performance variability. Similarly, online services seek to guarantee that a high percentile of their requests

complete within an acceptable amount of time [65, 107, 33]. Some of these works [114, 100] specifically target median performance; we go beyond all this prior work, by considering full-range performance that also accounts for long-latency tails.

Finally, a few prior works have proposed to manage performance by adjusting system configurations, e.g. [121, 120]. These systems either had no performance target or needed to satisfy a single-point service-level agreement (e.g., 99^{th} percentile performance lower than 100ms). OpTune differs from these efforts as it allows administrators to specify multiple performance targets.

An additional related topic is Web Service composition [20, 1]. While these works do consider QoS issues [40], we go beyond them by studying full-range performance tuning.

Chapter 3

A Capacity-Optimized SSD Cache for Primary Storage

3.1 Introduction

In this chapter, we present Nitro, an SSD cache that applies advanced data reduction techniques to SSD caches, increasing the effective cache size and reducing SSD costs for a given system. Deduplication (replacing a repeated data block with a reference) and compression (e.g. LZ) of storage have become the primary strategies to achieve high space and energy efficiency, with most research performed on HDD systems. We refer to the combination of deduplication and compression for storage as capacity-optimized storage (COS), which we contrast with traditional primary storage (TPS) without such features.

Though deduplicating SSDs [24, 60] and compressing SSDs [49, 75, 115] has been studied independently, using both techniques in combination for caching introduces new complexities. Unlike the variable-sized output of compression, the Flash Translation Layer (FTL) supports page reads (e.g. 8KB). The multiple references introduced with deduplication conflicts with SSD erasures that take place at the block level (a group of pages, e.g. 2MB), because individual pages of data may be referenced while the rest of a block could otherwise be reclaimed. Given the high churn of a cache and the limited erase cycles of SSDs, our technique must balance performance concerns with the limited lifespan of SSDs. We believe this is the first study combining deduplication and compression to achieve capacity-optimized SSDs.

Our design is motivated by an analysis of deduplication patterns of primary storage traces and properties of local compression. Primary storage workloads vary in how frequently similar

content is accessed, and we wish to minimize deduplication overheads such as in-memory indices. For example, related virtual machines (VMs) have high deduplication whereas database logs tend to have lower deduplication, so Nitro supports targeting deduplication where it can have the most benefit. Since compression creates variable-length data, Nitro packs compressed data into larger units, called Write-Evict Units (WEUs), which align with SSD internal blocks. To extend SSD lifespan, we chose a cache replacement policy that tracks the status of WEUs instead of compressed data, which reduces SSD erasures. An important finding is that replacing WEUs instead of small data blocks maintains nearly the same cache hit ratio and performance of finer-grained replacement, while extending SSD lifespan.

To evaluate Nitro, we developed and validated a simulator and two prototypes. The prototypes place Nitro in front of commercially available storage products. The first prototype uses a cos system with deduplication and compression. The system is typically targeted for storing highly redundant, sequential backups. Therefore, it has lower random I/O performance, but it becomes a plausible primary storage system with Nitro acceleration. The second prototype uses a TPS system without deduplication or compression, which Nitro also accelerates.

Because of the limited computational power and memory of SSDs [60] and to facilitate the use of off-the-shelf SSDs, our prototype implements deduplication and compression in a layer above the SSD FTL. Our evaluation demonstrates that Nitro improves I/O performance because it can service a large fraction of read requests from an SSD cache with low overheads. It also illustrates the trade-offs between performance, RAM, and SSD lifespan. Experiments with prototype systems demonstrate additional benefits including improved random read performance in aged systems, faster snapshot restore when snapshots overlap with primary versions in a cache, and reduced writes to SSDs because of duplicate content. In summary, our contributions are:

Summary of contributions. We propose Nitro, an SSD cache that utilizes deduplication,

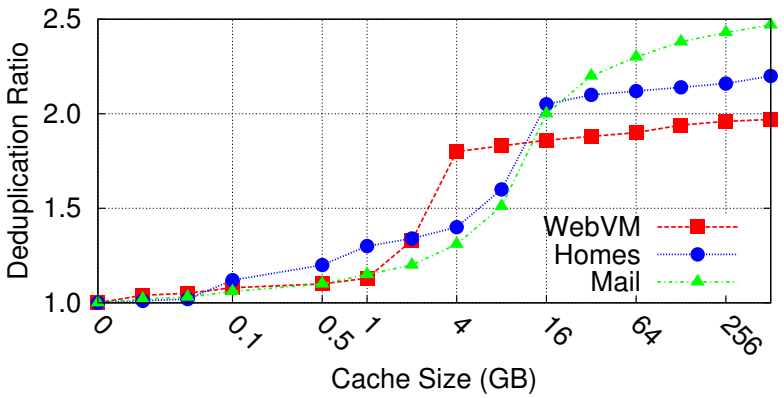


Figure 3.1: Caching tens of thousand of blocks will achieve most of the potential deduplication.

compression, and large replacement units to accelerate primary I/O. We investigate the trade-offs between deduplication, compression, RAM requirements, performance, and SSD lifespan. We experiment with both COS and TPS prototypes to validate Nitro's performance improvements.

The remainder of the paper proceeds as follows. The next section motivates our work and discusses some of the background. Section 3.3 describes Nitro and its architecture. Section 3.4 describes an SSD co-design to validate the benefits of WEU caching and our prototypes. Section 3.5 describes our storage traces and platform, and Section 3.6 presents our evaluation results.

3.2 Background and Discussion

In this section, we discuss the potential benefits of adding deduplication and compression to an SSD cache and then discuss the appropriate storage layer to add a cache.

Leveraging duplicate content in a cache. I/O rates for primary storage can be accelerated if data regions with different addresses but duplicate content can be reused in a cache. While previous work focused on memory caching and replicating commonly used data to minimize disk seek times [63], we focus on SSD caching.

We analyzed storage traces (described in §3.5) to understand opportunities to identify repeated content. Figure 3.1 shows the deduplication ratio (defined in §3.5.1) for 4KB blocks for various cache sizes. The deduplication ratios increase slowly for small caches and then grow rapidly to $\sim 2.0X$ when the cache is sufficiently large to hold the working set of unique content. This result confirms that a cache has the potential to capture a significant fraction of potential deduplication [60].

This result motivates our efforts to build a deduplicated SSD cache to accelerate primary storage. Adding deduplication to a storage system increases complexity, though, since infrastructure is needed to track the liveness of blocks. In contrast, caching requires less complexity, since cache misses do not cause a data loss for write-through caching, though performance is affected. Also, the overhead of calculating and managing secure fingerprints must not degrade overall performance.

Leveraging compression in a cache. Compression, like deduplication, has the potential to increase cache capacity. Previous studies [29, 49, 105, 115] have shown that local compression saves from 10-60% of capacity, with an approximate mean of 50% using a fast compressor such as LZ. Potentially doubling our cache size is desirable, as long as compression and decompression overheads do not significantly increase latency. Using an LZ-style compressor is promising for a cache, as compared to a HDD system that might use a slower compressor that achieves higher compression. Decompression speed is also critical to achieve low latency storage, so we compress individual data blocks instead of concatenating multiple data blocks before compression. Our implementation has multiple compression/decompression threads, which can leverage future advances in multi-core systems.

A complexity of using compression is that it transforms fixed-sized blocks into variable-sized blocks, which is at odds with the properties of SSDs. Similar to previous work [49, 75, 115], we pack compressed data together into larger units (WEUs). Our contribution focuses on exploring the caching impact of these large units, which achieves compression benefits while

decreasing SSD erasures.

Appropriate storage layer for Nitro. Caches have been added at nearly every layer of storage systems: from client-side caches to the server-side, and from the protocol layer (*e.g.* NFS) down to caching within hard drives. For a deduplicated and compressed cache, we believe there are two main locations for a server-side cache. The first is at the highest layer of the storage stack, right after processing the storage protocol. This is the server's first opportunity to cache data, and it is as close to the client as possible, which minimizes latency.

The second location to consider is post-deduplication (and compression) within the system. The advantage of the post-deduplication layer is that currently existing functionality can be reused. Of course, deduplication and compression have not yet achieved wide-spread implementation in storage systems. An issue with adding a cache at the post-deduplication layer is that some mechanism must provide the file recipe, a structure mapping from file and offset to fingerprint (*e.g.* SHA-1 hash of data), for every cache read. Loading file recipes adds additional I/O and latency to the system, depending on the implementation. While we added Nitro at the protocol layer, for cos systems, we evaluate the impact of using file recipes to accelerate duplicate reads (Section 3.3.1). We then compare to TPS systems that do not typically have file recipes, but do benefit from caching at the protocol layer.

3.3 Nitro Architecture and Design

This section presents the design of our Nitro architecture. Starting at the bottom of Figure 3.2, we use either cos or TPS HDD systems for large capacity. The middle of the figure shows SSDs used to accelerate performance, and the upper layer shows in-memory structures for managing the SSD and memory caches.

Nitro is conceptually divided into two halves shown in Figure 3.2 and in more detail in Figure 3.3 (steps 1-6 are described in §3.3.2). The top half is called the *CacheManager*, which manages the cache infrastructure (indices), and a lower half that implements SSD caching. The

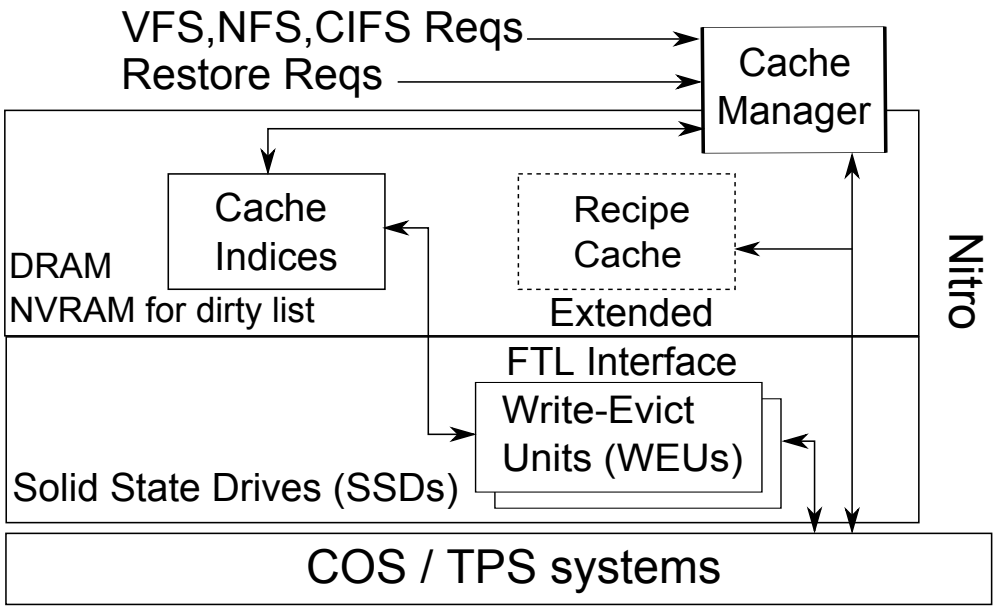


Figure 3.2: SSD cache and disk storage.

CacheManager maintains a file index that maps the file system interface (<filehandle, offset>) to internal SSD locations; a fingerprint index that detects duplicate content before it is written to SSD; and a dirty list that tracks dirty data for write-back mode. While our description focuses on file systems, other storage abstractions such as volumes or devices are supported. The CacheManager is the same for our simulator and prototype implementations, while the layers below differ to either use simulated or physical SSDs and HDDs (§5.2.4).

We place a small amount of NVRAM in front of our cache to buffer pending writes and to support write-back caching: check-pointing and journaling of the dirty list. The CacheManager implements a dynamic prefetching scheme that detects sequential accesses when the consecutive bytes accessed metric (M11 in [69]) is higher than a threshold across multiple-streams. Our cache is scan-resistant because prefetched data that is accessed only once in memory will not be cached. We currently do not cache file system metadata because we do not expect it to deduplicate or compress well, and we leave further analysis to future work.

3.3.1 Nitro Components

Extent. An extent is the basic unit of data from a file that is stored in the cache, and the cache indices reference extents that are compressed and stored in the SSDs. We performed a large number of experiments to size our extents, and there are trade-offs in terms of read-hit ratio, SSD erasures, deduplication ratio, and RAM overheads. As one example, smaller extents capture finer-grained changes, which typically results in higher deduplication ratios, but smaller extents require more RAM to index. We use the median I/O size of the traces we studied (8KB) as the default extent size. For workloads that have differing deduplication and I/O patterns than what we have studied, a different extent size (or dynamic sizing) may be more appropriate.

Write-Evict Unit (WEU). The Write-Evict Unit is our unit of replacement (writing and evicting) for SSD. File extents are compressed and packed together into one WEU in RAM, which is written to an SSD when it is full. Extents never span WEUs. We set the WEU size equal to one or multiple SSD block(s) (the unit for SSD erase operation) depending on internal SSD properties, to maximize parallelism and reduce internal fragmentation. We store multiple file extents in a WEU. Each WEU has a header section describing its contents, which is used to accelerate rebuilding the RAM indices at start-up. The granularity of cache replacement is an entire WEU, thus eliminating copy forward of live-data to other physical blocks during SSD garbage collection (GC). This replacement strategy has the property of reducing erasures within an SSD, but this decision impacts performance, as we discuss extensively in §3.6.1. WEUs have generation numbers indicating how often they have been replaced, which are used for consistency checks as described later.

File index. The file index contains a mapping from filehandle and offset to an extent's location in a WEU. The location consists of the WEU ID number, the offset within the WEU, and the amount of compressed data to read. Multiple file index entries may reference the same extent due to deduplication. Entries may also be marked as dirty if write-back mode is supported (shown in gray in Figure 3.3).

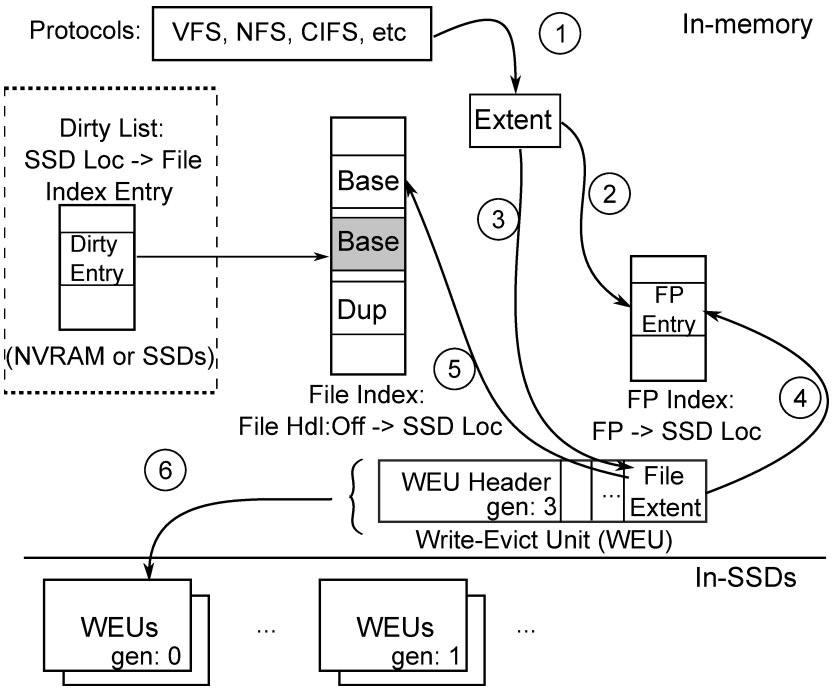


Figure 3.3: File index with base and duplicate entries, fingerprint index, file extents stored in WEUs, and a dirty list.

Fingerprint index. To implement deduplication within the SSDs, we use a fingerprint index that maps from extent fingerprint to an extent's location within the SSD. The fingerprint index allows us to find duplicate entries and effectively increase the cache capacity. Since primary workloads may have a wide range of content redundancy, the fingerprint index size can be limited to any arbitrary level, which allows us to make trade-offs between RAM requirements and how much potential deduplication is discovered. We refer to this as the *fingerprint index ratio*, which creates a partial fingerprint index. For a partial fingerprint index, a policy is needed to decide which extents should be inserted into/evicted from the fingerprint index. User-specified configurations, folder/file properties, or access patterns could be used in future work. We currently use LRU eviction, which performed as well as more complicated policies.

Recipe cache. To reduce misses on the read-path, we create a cache of file recipes (Figure 3.2), which represent a file as a sequence of fingerprints referencing extents. This allows us to check

the fingerprint index for already cached, duplicate extents. File recipes are a standard component of cos systems and can be prefetched to our cache, though this requires support from the cos system. Since fingerprints are small (20 bytes) relative to the extent size (KBs), prefetching large lists of fingerprints in the background can be efficient compared to reading the corresponding data from HDD storage. A recipe cache can be an add-on for TPS to opportunistically improve read performance. We do not include a recipe cache in our current TPS implementation because we want to isolate the impact of Nitro without changing other properties of the underlying systems. Its impact on performance is discussed in §3.6.2.

Dirty list. The CacheManager supports both write-through and write-back mode. Write-through mode assumes all data in the cache are clean because writes to the system are acknowledged when they are stable both in SSD and the underlying storage system. In contrast, write-back mode treats writes as complete when data are cached either in the NVRAM or SSD. In write-back mode, a dirty list tracks dirty extents, which have not yet propagated to the underlying disk system. The dirty list can be maintained in NVRAM (or SSD) for consistent logging since it is a compact list of extent locations. Dirty extents are written to the underlying storage system either when they are evicted from the SSD or when the dirty list reaches a size watermark. When a dirty file index entry is evicted (base or duplicate), the file recipe is also updated. The CacheManager then marks the corresponding file index entries as clean and removes the dirty list entries.

3.3.2 Nitro Functionality

File read path. Read requests check the file index based on filehandle and offset. If there is a hit in the file index, the CacheManager will read the compressed extent from a WEU and decompress it. The LRU status for the WEU is updated accordingly. For base entries found in the file index, reading the extent's header from SSD can confirm the validity of the extent. When reading a duplicate entry, the CacheManager confirms the validity with WEU generation

numbers. An auxiliary structure tracks whether each WEU is currently in memory or in SSD.

If there is a file index miss and the underlying storage system supports file recipes (i.e. cos), the CacheManager prefeteches the file recipe into the recipe cache. Subsequent read requests reference the recipe cache to access fingerprints, which are checked against the cache fingerprint index. If the fingerprint is found to be a duplicate, then cached data can be returned, thus avoiding a substantial fraction of potential disk accesses. The CacheManager updates the LRU status for the fingerprint index if there is a hit. If a read request misses in both the file and fingerprint indices, then the read is serviced from the underlying HDD system, returned to the client, and passed to the cache insertion path.

File write path. On a write, extents are buffered in NVRAM and passed to the CacheManager for asynchronous SSD caching.

Cache insertion path. To demonstrate the process of inserting an extent into the cache and deduplication, consider the following 6-step walk-through example in Figure 3.3: (1) Hash a new extent (either from caching a read miss or from the file write path) to create a fingerprint. (2) Check the fingerprint against the fingerprint index. If the fingerprint is in the index, update the appropriate LRU status and go to step 5. Otherwise continue with step 3. (3) Compress and append the extent to a WEU that is in-memory, and update the WEU header. (4) Update the fingerprint index to map from a fingerprint to WEU location. (5) Update the file index to map from file handle and offset to WEU. The first entry for the cached extent is marked as a “Base” entry. Note that the WEU header only tracks the base entry. (6) When an in-memory WEU becomes full, increment the generation number and write it to the SSD. In write-back mode, dirty extents and clean extents are segregated into separate WEUs to simplify eviction, and the dirty-list is updated when a WEU is migrated to SSD.

Handling of duplicate entries is slightly more complicated. Once a WEU is stored in SSD, we do not update its header because of the erase penalty involved. When a write consists of duplicate content, as determined by the fingerprint index, a duplicate entry is created in the file

index (marked as “Dup”) which points to the extent’s location in SSD WEU. Note that when a file extent is over-written, the file index entry is updated to refer to the newest version. Previous version(s) in the SSD may still be referenced by duplicate entries in the file index.

SSD cache replacement policy. Our cache replacement policy selects a WEU from the SSD to evict before reusing that space for a newly packed WEU. The CacheManager initiates cache replacement by migrating dirty data from the selected WEU to disk storage and removing corresponding invalid entries from the file and fingerprint indices. To understand the interaction of WEU and SSDs, we experimented with moving the cache replacement decisions to the SSD, on the premise that the SSD FTL has more internal knowledge. In our co-designed SSD version (§5.2.4), the CacheManager will query the SSD to determine which WEU should be replaced based on recency. If the WEU contains dirty data, the CacheManager will read the WEU and write dirty extents to underlying disk storage.

Cleaning the file index. When evicting a WEU from SSD, our in-memory indices must also be updated. The WEU metadata allows us to remove many file index entries. It is impractical, though, to record back pointers for all duplicate entries in the SSD, because these duplicates may be read/written hours or days after the extent is first written to a WEU. Updating a WEU header with a back pointer would increase SSD churn. Instead, we use asynchronous cleaning to remove invalid, duplicate file index entries. A background cleaning thread checks all duplicate entries and determines whether their generation number matches the WEU generation number. If a stale entry is accessed by a client before it is cleaned, then a generation number mismatch indicates that the entry can be removed. All of the WEU generation numbers can be kept in memory, so these checks are quick, and rollover cases are handled.

Faster snapshot restore/access. Nitro not only accelerates random I/Os but also enables faster restore and/or access of snapshots. The SSD can cache snapshot data as well as primary data for cos storage, distinguished by separate snapshot file handles.

We use the standard snapshot functionality of the storage system in combination with file

recipes for cos. When reading a snapshot, its recipe will be prefetched from disk into a recipe cache. Using the fingerprint index, duplicate reads will access extents already in the cache, so any shared extents between the primary and snapshot versions can be reused, without additional disk I/O. To accelerate snapshot restores for TPS, integration with differential snapshot tracking is needed.

System restart. Our cache contains numerous extents used to accelerate I/O, and warming up a cache after a system outage (planned or unplanned) could take many hours. To accelerate cache warming, we implemented a system restart/crash recovery technique [93]. A journal tracks the dirty and invalid status of extents. When recovering from a crash, the CacheManager reads the journal, the WEU headers from SSD (faster than reading all extent headers), and recreates indices. Note that our restart algorithm only handles base entries and duplicate entries that reference dirty extents (in write-back mode). Duplicate entries for clean extents are not explicitly referenced from WEU headers, but they can be recovered efficiently by fingerprint lookup when accessed by a client, with only minimal disk I/O to load file recipes.

3.4 Nitro Implementation

To evaluate Nitro, we developed a simulator and two prototypes. The CacheManager is shared between implementations, while the storage components differ. Our simulator measures read-hit ratios and SSD churn, and its disk stub generates synthetic content based on fingerprint. Our prototypes measure performance and use real SSDs and HDDs.

Potential SSD customization. Most of our experiments use standard SSDs without any modifications, but it is important to validate our design choices against alternatives that modify SSD functionality. Previous projects [7, 21, 60] showed that the design space of the FTL can lead to diverse SSD characteristics, so we would like to understand how Nitro would be affected by potential SSD changes. Interestingly, we found through simulation that Nitro performs nearly as well with a commercial SSD as with a customized SSD.

We explored two FTL modifications, as well as changes to the standard GC algorithm. First, the FTL needs to support aligned allocation of contiguous physical pages for a WEU across multiple planes in aligned blocks, similar to vertical and horizontal super-page striping [21]. Second, to quantify the best-case of using SSD as a cache, we push the cache replacement functionality to the FTL, since the FTL has perfect information about page state. Thus, a new interface allows the CacheManager to update indices and implement write-back mode before eviction. We experimented with multiple variants and present WEU-LRU, an update to the greedy SSD GC algorithm that replaces WEUs.

We also added the SATA TRIM command [102] in our simulator, which invalidates a range of SSD logical addresses. When the CacheManager issues TRIM commands, the SSD performs GC without copying forward data. Our SSD simulator is based on well-studied simulators [7, 21] with a hybrid mapping scheme [66] where blocks are categorized into data and log blocks. Page-mapped log blocks will be consolidated into block-mapped data blocks through merge operations. Log blocks are further segregated into sequential regions and random areas to reduce expensive merge operations.

Prototype system. We have implemented a prototype Nitro system in user space, leveraging multi-threading and asynchronous I/O to increase parallelism and with support for replaying storage traces. We use real SSDs for our cache, and either a COS or TPS system with hard drives for storage (§ 3.5). We confirmed the cache hit ratios are the same between the simulator and prototypes. When evicting dirty extents from SSD, they are moved to a write queue and written to disk storage before their corresponding WEU is replaced.

3.5 Experimental Methodology

In this section, we first describe our analysis metrics. Second, we describe several storage traces used in our experiments. Third, we discuss the range of parameters explored in our evaluation. Fourth, we present the platform for our simulator and prototype systems.

3.5.1 Metrics

Our results present overall system IOPS, including both reads and writes. Because writes are handled asynchronously and are protected by NVRAM, we further focus on read-hit ratio and read response time to validate Nitro. The principal evaluation metrics are:

IOPS: Input/Output operations per second.

Read-hit ratio: The ratio of read I/O requests satisfied by Nitro over total read requests.

Read response time (RRT): The average elapsed time from the dispatch of one read request to when it finishes, characterizing the user-perceivable latency.

SSD erasures: The number of SSD blocks erased, which counts against SSD lifespan.

Deduplication and compression ratios: Ratio of the data size versus the size after deduplication or compression ($\geq$ 1X). Higher values indicate more space savings.

3.5.2 Experimental Traces

Most of our experiments are with real-world traces, but we also use synthetic traces to study specific topics.

FIU traces: Florida International University (FIU) collected storage traces across multiple weeks, including WebVM (a VM running two web-servers), Mail (an email server with small I/Os), and Homes (a file server with a large fraction of random writes). The FIU traces contain content fingerprint information with small granularity (4KB or 512B), suitable for various extent size studies. The FIU storage systems were reasonably sized, but only a small region of the file systems was accessed during the trace period. For example, WebVM, Homes and Mail have file system sizes of 70GB, 470GB and 500GB in size, respectively, but we measured that the traces only accessed 5.3%, 5.8% and 11.5% of the storage space, respectively [63]. The traces have more writes than reads, with write-to-read ratios of 3.6, 4.2, and 4.3, respectively. To our knowledge, the FIU traces are the only publicly available traces with content.

Boot-storm trace: A “boot-storm” trace refers to many VMs booting up within a short time

frame from the same storage system [38]. We first collected a trace while booting up one 18MB VM kernel in Xen hypervisor. The trace consisted of 99% read requests, 14% random I/O, and 1.2X deduplication ratio. With this template, we synthetically produced multiple VM traces in a controlled manner representing a large number of cloned VMs with light changes. Content overlap was set at 90% between VMs, and the addresses of duplicates were shifted by 0-15% of the address space.

Restore trace: To study snapshot restore, we collected 100 daily snapshots of a 38GB workstation VM with a median over-write rate of 2.3%. Large read I/Os (512KB) were issued while restoring the entire VM.

Fingerprint generation. The FIU traces only contain fingerprints for one block size (e.g. 4KB), and we want to vary the extent size for experiments (4-128KB), so it is necessary to process the traces to generate extent fingerprints. We use a multi-pass algorithm, which we briefly describe. The first pass records the fingerprints for each block read in the trace, which is the initial state of the file system. The second pass replays the trace and creates extent fingerprints. An extent fingerprint is generated by calculating a SHA-1 hash of the concatenated block fingerprints within an extent, filling in unspecified block fingerprints with unique values as necessary. Write I/Os within the trace cause an update to block fingerprints and corresponding extent fingerprints. A final pass replays the modified trace for a given experiment.

Synthetic compression information. Since the FIU traces do not have compression information, we synthetically generate content with intermingled unique and repeated data based on a compression ratio parameter. Unless noted, the compression ratio is set for each extent using a normal distribution with mean of 2 and variance of 0.25, representing a typical compression ratio for primary workloads [105]. We used LZ4 [44] for compression and decompression in the prototype.

Variable	Values
Fingerprint index ratio (%)	**100**, 75, 50, 25, 0 (off)
Compression	**on**, off
Extent size (KB)	4, **8**, 16, 32, 64, 128
Write/Evict granularity	**WEU**, extent
Cache size (% of volume)	0.5, 1, **2**, 5
WEU size (MB)	0.5, 1, **2**, 4
Co-design	**standard SSD**, modified SSD
Write-mode	write-through, **write-back**
Prefetching	dynamic up to 128KB
Backend storage	cos, TPS

Table 3.1: Parameters for Nitro with default values in bold.

3.5.3 Parameter Space

Table 3.1 lists the configuration space for Nitro, with default values in bold. Due to space constraints, we interleave parameter discussion with experiments in the evaluation section. While we would like to compare the impact of compression using WEU-caching versus plain extent-based caching, it is unclear how to efficiently store compressed (variable-sized) extents to SSDs without using WEUs or an equivalent structure [49, 75, 115]. For that reason, we show extent caching without compression, but with or without deduplication, depending on the experiment. The cache is sized as a fraction of the storage system size. For the FIU traces, a 2% cache corresponds to 1.4GB, 9.4GB, and 10GB for WebVM, Homes and Mail traces respectively. Most evaluations are with the standard SSD interface except for a co-design evaluation. We use the notation Deduplicated (D), Non-deduplicated (ND), Compressed (C) and Non-compressed (NC). Nitro uses the WEU (D, C) configuration by default.

3.5.4 Experimental Platform

Our prototype with a cos system is a server equipped with 2.33GHz Xeon CPUs (two sockets, each with two cores supporting two hardware threads). The system has 36GB of DRAM, 960MB of NVRAM, and two shelves of hard drives. One shelf has 12 1TB 7200RPM SATA hard drives, and the other shelf has 15 7200RPM 2TB drives. Each shelf has a RAID-6 configuration including two spare disks. For comparison, the TPS system is a server equipped with four

1.6GHz Xeon CPUs and 8GB DRAM with battery protection. There are 11 1TB 7200RPM disk drives in a RAID-5 configuration. Before each run, we reset the initial state of the HDD storage based on our traces.

Both prototypes use a Samsung 256GB SSD, though our experiments use a small fraction of the available SSD, as controlled by the cache capacity parameter. According to specifications, the SSD supports >100K random read IOPS and >90K random write IOPS. Using a SATA-2 controller (3.0 Gbps), we measured 8KB SSD random reads and writes at 18.7K and 4.2K IOPS, respectively. We cleared the SSD between experiments.

We set the SSD simulation parameters based on the Micron MLC SSD specification [83]. We vary the size of each block or flash chip to control the SSD capacity. Note that a larger SSD block size has longer erase time (e.g., 2ms for 128KB and 3.8ms for 2MB). For the unmodified SSD simulation, we over-provision the SSD capacity by 7% for garbage collection, and we reserve 10% for log blocks for the hybrid mapping scheme. No space reservation is used for the modified SSD WEU variants.

3.6 Evaluation

This section presents our experimental results. We first measure the impact of deduplication and compression on caching as well as techniques to reduce in-memory indices and to extend SSD lifespan. Second, we evaluate Nitro performance on both COS and TPS prototype systems and perform sensitivity and overhead analysis. Finally, we study Nitro's additional advantages.

3.6.1 Simulation Results

We start with simulation results, which demonstrate caching improvements with deduplication and compression and compare a standard SSD against a co-design that modifies an SSD to specifically support caching.

Read-hit ratio. We begin by showing Nitro's effectiveness at improving the read-hit ratio,

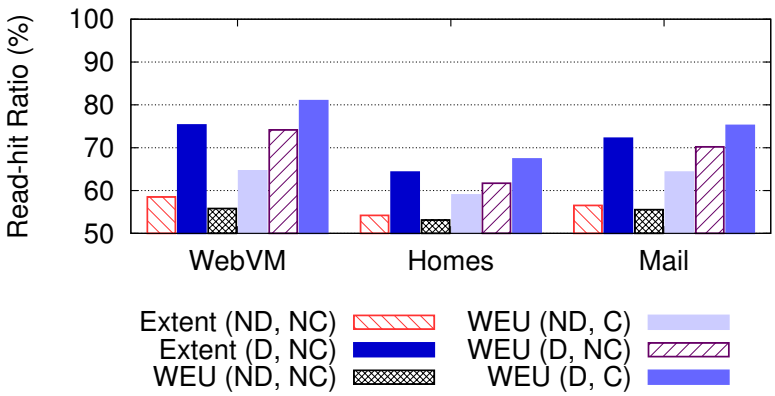
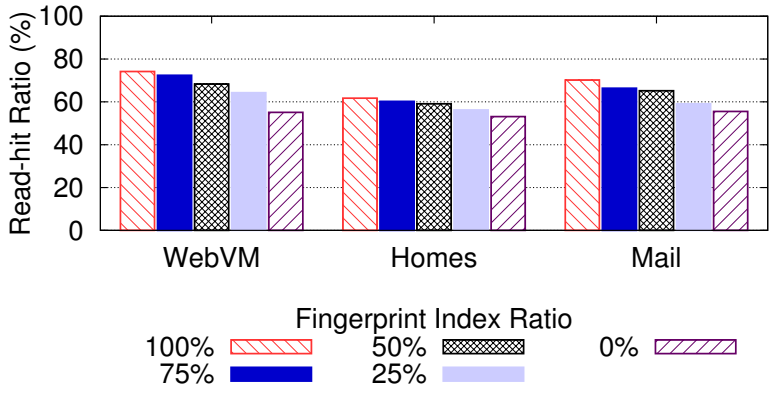


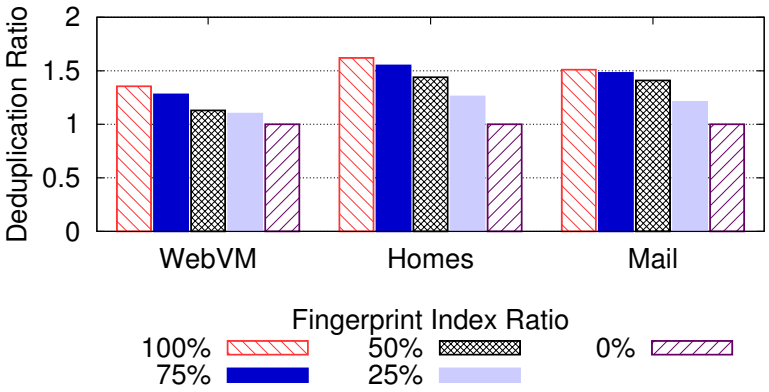
Figure 3.4: Read-hit ratio of WEU-based vs. Extent-based for all workloads. Y-axis starts at 50%.

which is shown in Figure 3.4 for all three FIU traces. The trend for all traces is that adding deduplication and compression increases the read-hit ratio.

WEU (D, C) with deduplication (fingerprint index ratio set to 100% of available SSD extent entries) and compression represents the best scenario with improvements of 25%, 14% and 20% across all FIU traces as compared to a version without deduplication or compression (WEU (ND, NC)). Adding compression increases the read-hit ratio for WEU by 5-9%, and adding deduplication increases the read-hit ratio for WEU by 8-19% and extents by 6-17%. Adding deduplication consistently offers a greater improvement than adding compression, suggesting deduplication is capable of increasing the read-hit ratio for primary workloads that contain many duplicates like the FIU traces. Comparing WEU and extent-based caching with deduplication, but without compression (D, NC), extent-based caching has a slightly higher hit-ratio by 1-4% due to finer-grained evictions. However, the advantages of extent-based caching are offset by increased SSD erasures, which are presented later. In an experiment that increased the cache size up to 5% of the file system size, the combination of deduplication and compression (D, C) showed the largest improvement. These results suggest Nitro can extend the caching benefits of SSDs to much larger disk storage systems.



(a) Impact of fingerprint index ratio on read-hit ratio



(b) Impact of fingerprint index ratio on deduplication ratio

Figure 3.5: Fingerprint index ratio impact on read-hit ratio and deduplication for WEU (D, NC).

Impact of fingerprint index ratio. To study the impact of deduplication, we adjust the fingerprint index ratio for WEU (D, NC). 100% means that all potential duplicates are represented in the index, while 0% means deduplication is turned off. Decreasing the fingerprint index ratio directly reduces the RAM footprint (29 bytes per entry) but also likely decreases the read-hit ratio as the deduplication ratio drops.

Figure 3.5(a) shows the read-hit ratio drops gradually as the fingerprint index ratio decreases. Figure 3.5(b) shows that the deduplication ratio also slowly decreases with the fingerprint index ratio. Homes and Mail have higher deduplication ratios (≥1.5X) than WebVM, as shown in Figure 3.1. Interestingly, higher deduplication ratios in the Homes and Mail traces do not directly translate to higher read-hit ratios because there are more writes than reads (~4 W/R ratio), but do increase IOPS (§3.6.2). Nitro users could limit their RAM footprint by setting the

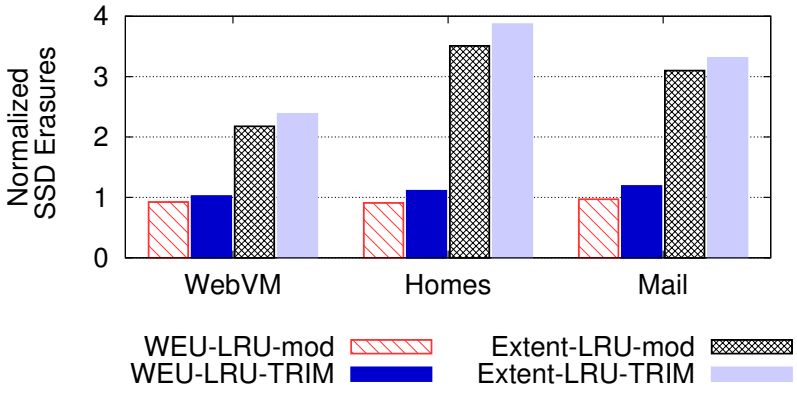


Figure 3.6: Number of SSD erasures for modified and unmodified SSD variants.

fingerprint index ratio to 75% or 50%, which results in a 16-22% RAM savings respectively and a decrease in read-hit ratio of 5-11%. For example, when reducing the fingerprint index from 100% to 50% for the Mail trace (10GB cache size), ≥131,000 duplicate extents are not cached by Nitro, on average.

WEU vs. SSD co-design. So far, we considered scenarios where the SSD is unmodified. Next we compare our current design to an alternative that modifies an SSD to directly support WEU caching. In this experiment, we study the impact of silent eviction and the WEU-LRU eviction policy (discussed in §5.2.4) on SSD erasures. Our co-design specifically aligns WEUs to SSD blocks (WEU-LRU-mod). We also compare our co-design to variants using the TRIM command (WEU/extent-LRU-TRIM), which alerts the FTL that a range of logical addresses can be released. Figure 3.6 plots SSD erasures normalized relative to WEU-LRU without SSD modifications (1.0 on the vertical axis) and compares WEU versus extent caching.

SSD erasures are 2-4X higher for the extent-LRU-mod approach (i.e. FlashTier [93] extended to use an LRU policy) and extent-LRU-TRIM approach as compared to both WEU versions. This is because the CacheManager lacks SSD layout information so that extent-based eviction cannot completely avoid copying forward live SSD data. Interestingly, utilizing TRIM with the WEU-LRU-TRIM approach has similar results to WEU-LRU-mod, which indicates

the CacheManager could issue TRIM commands before overwriting WEUs instead of modifying the SSD interface. We also analyzed the impact of eviction policy on read-hit ratio. WEU-LRU-mod achieves a 5-8% improvement in read-hit ratio compared to an unmodified version across the FIU traces.

Depending on the data set, the number of SSD erasures varied for the FTL and TRIM alternatives, with results between 9% fewer and 20% more erasures than using WEUs. So, using WEUs for caching is a strong alternative when it is impractical to modify the SSD or when the TRIM command is unsupported. Though not shown, caching extents without SSD modifications or TRIM (naive caching) resulted in several orders of magnitude more erasures than using WEUs.

3.6.2 Prototype System Results

Next, we report the performance of Nitro for primary workloads on both COS and TPS systems. We then present sensitivity and overheads analysis of Nitro. Note that the cache size for each workload is 2% of the file system size for each dataset unless otherwise stated.

Performance in COS **system.** We first show how a high read-hit ratio in our Nitro prototype translates to an overall performance boost. We replayed the FIU traces at an accelerated speed to use ~95% of the system resources, (reserving 5% for background tasks), representing a sustainable high load that Nitro can handle. We setup a warm cache scenario where we use the first 16 days to warm the cache and then measure the performance for the following 5 days.

Table 3.2 lists the improvement of total IOPS (reads and writes), and read response time reduction relative to a system without an SSD cache for all FIU traces. For example, a decrease in read response time from 4ms to 1ms implies a 75% reduction. For all traces, IOPS improvement is ≥254%, and the read response time reduction is ≥49% for Nitro WEU variants. In contrast, the Extent (ND, NC) column shows a baseline SSD caching system without the benefit of deduplication, compression, or WEU. The read-hit ratio is consistent with Figure 3.4.

Metric (%)	***Trace***	***Extent***	***Nitro WEU Variants***			
		ND, NC	ND, NC	ND, C	D, NC	D, C
cos system						
IOPS	WebVM	251	307	393	532	661
	Homes	259	341	432	556	673
	Mail	213	254	292	320	450
RRT	WebVM	52	54	63	72	78
	Homes	46	49	55	57	62
	Mail	50	53	61	67	72
TPS system						
IOPS	WebVM	93	113	148	198	264
	Homes	90	130	175	233	287
	Mail	56	75	115	122	165
RRT	WebVM	39	41	49	58	64
	Homes	39	42	47	49	54
	Mail	41	44	51	57	61

Table 3.2: Performance evaluation of Nitro and its variants. We report IOPS improvement and read response time (RRT) reduction percentage relative to cos and TPS systems without an SSD cache. The standard deviation is ≤7.5%.

We observe that with deduplication enabled (D, NC), our system achieves consistently higher IOPS compared to the compression-only version (ND, C). This is because finding duplicates in the SSD prevents expensive disk storage accesses, which have a larger impact than caching more data due to compression. Nitro (D, C) achieves the highest IOPS improvement (673%) in Homes using cos. As explained before, a high deduplication ratio indicates that duplicate writes are canceled, which contributes to the improved IOPS. For Mail, the increase of deduplication relative to compression-only version is smaller because small I/Os (29% of I/Os are ≤ the 8KB extent size) can cause more reads from disk on the write path, thus negating some of the benefits of duplicate hits in the SSDs.

Compared to extent-based caching, WEU (D, C) improves non-normalized IOPS up to 120% and reduces read response time up to 55%. Compared to WEU (ND, NC), extent-based caching decreases IOPS 13-22% and increases read response time 4-7%. This is partially because extent-based caching increases the SSD write penalty due to small SSD overwrites. From SSD random write benchmarks, we found that 2MB writes (WEU size) have ~60% higher throughput than 8KB writes (extent size), demonstrating the value of large writes to SSD.

We also performed cold cache experiments that replay the trace from the last 5 days without

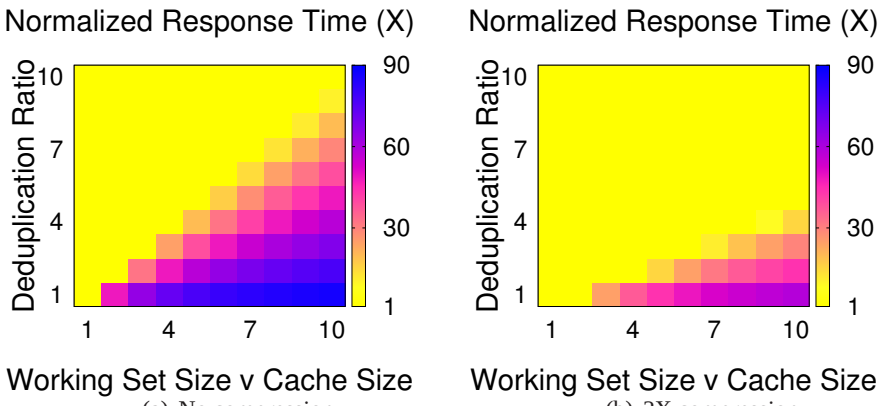
warming up Nitro. Nitro still improves IOPS up to 520% because of sequential WEU writes to the SSD. Read response time reductions are 2-29% for Nitro variants across all traces because fewer duplicated extents are cached in the SSD.

Performance in TPS system. Nitro also can benefit a TPS system (Table 3.2). Note that Nitro needs to compute extent fingerprints before performing deduplication, which is computation that can be reused in COS but not TPS. In addition, Nitro cannot leverage a recipe cache for TPS to accelerate read requests, which causes 5-14% loss in read hit-ratio for our WEU variants.

For all traces, the improvement of total IOPS (reads and writes) is ≥75%, and the read response time reduction is ≥41% for Nitro WEU variants. While deduplication and compression improve performance, the improvement across Nitro variants is lower relatively than for our COS system because storage systems without capacity-optimized techniques (e.g. deduplication and compression) have shorter processing paths, thus better baseline performance. For example, overwrites in existing deduplication systems can cause performance degradation because metadata updates need to propagate changes to an entire file recipe structure. For these reasons, the absolute IOPS is higher than COS with faster read response times. Cold cache results are consistent with warm cache results.

Sensitivity analysis. To further understand the impact of deduplication and compression on caching, we use synthetic traces to investigate the impact on random read performance, which represents the worst-case scenario for Nitro. Note that adding non-duplicate writes to the traces would equivalently decrease the cache size (e.g. multi-stream random reads and non-duplicate writes). Two parameters control the synthetic traces: (1) The ratio of working set size versus the cache size and (2) the deduplication ratio. We vary both parameters from 1 to 10, representing a large range of possible scenarios.

Figure 3.7 shows projected 2D contour graphs from a 3D plot for (D, NC) and (D, C). The metric is read response time in COS with Nitro normalized against that of fitting the entire data set in SSD (lower values are better). The horizontal axis is the ratio of working set size



(a) No compression (b) 2X compression

Figure 3.7: Sensitivity analysis of (D, NC) and (D, C).

versus cache size, and the vertical axis is the deduplication ratio. The bottom left corner (1, 1) is a working set that is the same size as the cache with no deduplication. We can derive the effective cache size from the compression and deduplication ratio. For example, the effective cache size for a 16GB cache in this experiment expands to 32GB with a 2X deduplication ratio configuration, and further to 64GB when adding 2X compression.

First, both deduplication and compression are effective techniques to improve read response time. For example, when the deduplication ratio is high (e.g. ≥5X such as for multiple, similar VMs), Nitro can achieve response times close to SSD even when the working set size is 5X larger than the cache size. The combination of deduplication and compression can support an even larger working set size. Second, when the deduplication ratio is low (e.g.≤2X), performance degrades when the working set size is greater than twice the cache size. Compression has limited ability to improve response time, and only a highly deduplicated scenario (e.g. VM boot-storm) can counter a large working set situation. Third, there is a sharp transition from high response time to low response time for both (D, NC) and (D, C) configurations (values jump from 1 to > 8), which indicates that (slower) disk storage has a greater impact on response time than (faster) SSDs. As discussed before, the performance for Nitro in the TPS system is always better than the cos system.

Nitro overheads. Figure 3.8 illustrates the performance overheads of Nitro with low and high

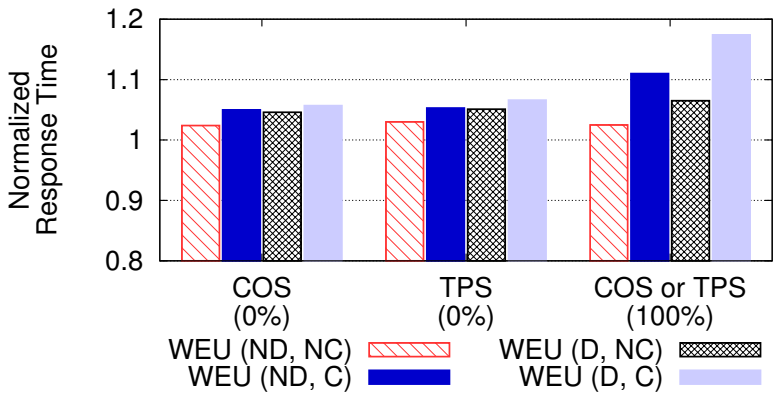


Figure 3.8: Overheads of Nitro prototypes with the cache sized to have 0% and 100% hit-ratios. Y-axis starts at 0.8. The standard dev. is ≤3.8% in all cases.

hit-ratios. We performed a boot-storm experiment using a real VM boot trace (§3.5) synthetically modified to create 60 VM traces. For the 59 synthetic versions, we set the content overlap ratio to 90%. We set the cache size to achieve 0% (0GB) and 100% (1.2GB) hit-ratios in the SSD cache. With these settings, we expect Nitro's performance to approach the performance of disk storage and SSD storage.

In both COS and TPS 0% hit-ratio configurations, we normalized against corresponding systems without SSDs. All WEU variants impose ≤7% overhead in response time because extent compression and fingerprint calculation are performed off the critical path. In the 100% hit-ratio scenario, we normalize against a system with all data fitting in SSD without WEUs. WEU (ND, NC) imposes a 2% increase in response time. Compression-only (ND, C) and deduplication-only (D, NC) impose 11% and 6.2% overhead on response time respectively. WEU (D, C) overhead (≤18%) mainly comes from decompression, which requires additional time when reading compressed extents from SSD. Although we are not focused on comparing compression algorithms, we did quantify that gzip achieves 23-47% more compression than LZ4 (our default), which improves the read-hit ratio, though decompression is 380% slower for gzip.

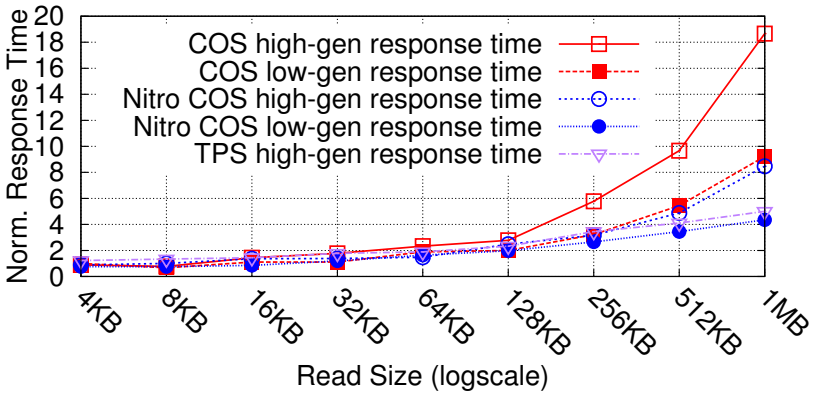


Figure 3.9: Response time diverges for random I/Os.

3.6.3 Nitro Advantages

There are additional benefits because Nitro effectively expands a cache: improved random read performance in aged cos, faster snapshot restore performance, and write reductions to SSD.

Random read performance in aged cos **system.** For HDD storage systems, unfortunately, deduplication can lead to storage fragmentation because a file's content may be scattered across the storage system. A previous study considered sequential reads from large backup files [73], while we study the primary storage case with random reads across a range of I/O sizes.

Specifically, we wrote 100 daily snapshots of a 38GB desktop VM to a standard cos system, a system augmented with the addition of a Nitro cache, and a TPS system. To age the system, we implemented a retention policy of 12 weeks to create a pattern of file writes and deletions. After writing each VM, we measured the time to perform 100 random reads for I/O sizes of 4KB to 1MB. The Nitro cache was sized to achieve a 50% hit ratio (19GB). Figure 3.9 shows timing results for the 1st generation (low-gen) and 100th generation (high-gen) normalized to the response time for cos low-gen at 4KB. For the TPS system, we only plot the high-gen numbers, which were similar to the low-gen results, since there was no deduplication-related fragmentation.

As the read size grows from 4KB to around 128KB, the response times are stable and the low-gen and high-gen results are close to each other for all systems. However, for larger read

sizes in the cos high-gen system, the response time grows rapidly. The cos system's logs indicate that the number of internal I/Os for the cos system is consistent with the high response times. In comparison to the cos system, the performance gap between low-gen and high-gen is smaller for Nitro. For 1MB random reads, Nitro cos high-gen response times (76ms) are slightly faster than cos low-gen, and Nitro cos low-gen response times (39ms) are slightly faster than a TPS high-gen system. By expanding an SSD cache, Nitro can reduce performance differences across random read sizes, though the impact of generational differences is not entirely removed.

Snapshot restore. Nitro can also improve the performance of restoring and accessing standard snapshots and clones, because of shared content with a cached primary version. Figure 3.10 plots the restore time for 100 daily snapshots of a 38GB VM (same sequence of snapshots as the previous test). The restore trace used 512KB read I/Os, which generate random HDD I/Os in an aged, cos system described above.

We reset the cache before each snapshot restore experiment to the state when the 100th snapshot is created. We evaluate the time for restoring each snapshot version and report the average for groups of 25 snapshots with the cache sized at either 2% or 5% of the 38GB volume. The standard deviation for each group was ≤7s. Group 1-25 has the oldest snapshots, and group 76-100 has the most recent. For all cache sizes, WEU (D, C) has consistently faster restore performance than a compression-only version (ND, C). For the oldest snapshot group (1-25) with a 5% cache size, WEU (D, C) achieves a shorter restore time (374s) when deduplication and compression are enabled as compared to the system with compression only (513s). The recent snapshot group averages 80% content overlap with the primary version, while the oldest group averages 20% content overlap, as plotted against the right axis. Clearly, deduplication assists Nitro in snapshot restore performance.

Reducing writes to SSD. Another important issue is how effective our techniques are at reducing SSD writes compared to an SSD cache without Nitro. SSDs do not support in-place

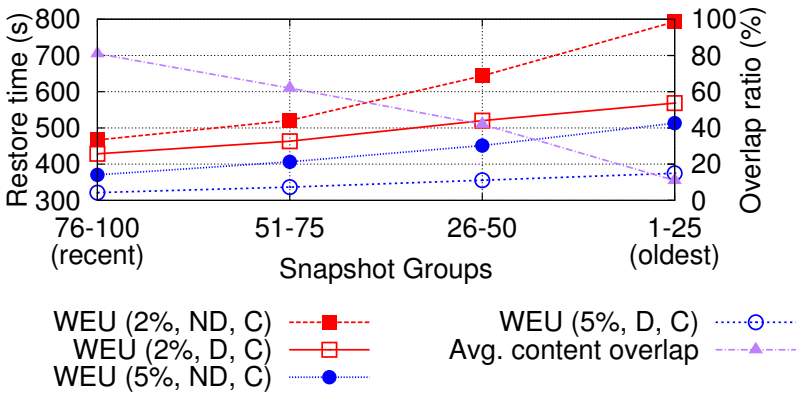


Figure 3.10: Nitro improves snapshot restore performance.

update, so deduplication can prevent churn for repeated content to the same, or a different, address. For WebVM and Mail, deduplication-only and compression-only reduces writes to SSD (≥22%), in which compression produces more savings compared to deduplication. In the Homes, deduplication reduces writes to SSD by 39% because of shorter fingerprint reuse distance. Deduplication and compression (D, C) reduces writes by 53%. Reducing SSD writes directly translates to extending the lifespan.

3.7 Summary

Nitro focuses on improving storage performance with a capacity-optimized SSD cache with deduplication and compression. To deduplicate our SSD cache, we present a fingerprint index that can be tuned to maintain deduplication while reducing RAM requirements. To support the variable-sized extents that result from compression, our architecture relies upon a Write-Evict Unit, which packs extents together and maximizes the cache hit-ratio while extending SSD lifespan. We analyze the impact of various design trade-offs involving cache size, fingerprint index size, RAM usage, and SSD erasures on overall performance. Extensive evaluation shows that Nitro can improve performance in both COS and TPS systems.

Chapter 4

I/O Predictability in Virtualized Multi-tenant Systems

4.1 Introduction

In this chapter, we address the specific case of *storage I/O* performance predictability (or simply I/O predictability). Compared to processing, memory, or networking, I/O predictability is the most challenging to achieve primarily due to the mechanical limitations of disk drives. In particular, high disk seek and rotational times make predictability difficult to achieve when multiple VMs share the same disk.

We divide the chapter into two parts. The first part quantifies the impact of workload characteristics, virtual machine monitor (VMM) architecture, the approach to virtualizing storage, and storage device characteristics on I/O predictability. Overall, we measure the I/O predictability of three workloads running on sixteen configurations. Using a new "performance deviation" (or simply "deviation") metric, we find widespread unpredictability. Further, perhaps surprisingly, we find that using an SSD as the storage device can exacerbate the problem when the workload is write-intensive.

Based on this first study, the second part of the chapter proposes and evaluates VirtualFence, a performance-predictable storage system. VirtualFence seeks to produce *consistent performance within the range defined by the best and worst performance levels that each VM may experience in a predictability-oblivious service* (i.e., in the absence of VirtualFence). Specifically, a VM would experience its best performance *BestPerf* when the provider co-locates no other VMs with it, and allows it to use all resources of the PM. The worst performance

WorstPerf would occur when the provider co-locates other active VMs with it (up to the maximum capacity of the PM). VirtualFence seeks to produce performance between *BestPerf* and *WorstPerf*. In addition, it seeks to retain this performance regardless of changes in the number of co-located VMs, changes in the working set size of the other VMs, or any other external condition. The goal is for each VM to always run as if it were alone on a fixed and tightly controlled partition of the resources.

To achieve this goal, VirtualFence couples a small persistent SSD cache with a much larger HDD. It also implements a non-work-conserving I/O scheduling algorithm, partitioning time into a fixed number of relatively coarse-grained slots. Each I/O slot can only be given to one active VM, but a single active VM may receive multiple slots (the number of slots depends on how much the VM's owner is willing to pay the cloud provider). I/O accesses from a VM are only serviced during the VM's assigned slot(s). A static allocation of time slots while a VM is active ensures consistent resource allocation for predictable performance. The SSD cache and I/O time partitioning together minimize the impact of HDD head movement due to consolidation, whereas I/O time partitioning minimizes the impact of SSD block erasures. Finally, VirtualFence partitions both the SSD cache and the HDD, which is a non-work-conserving space allocation scheme that again ensures consistent resource allocation for predictability.

For simplicity, we implement VirtualFence for a virtualized system with direct-attached disks. However, the same ideas (one SSD per HDD, SSD space partitioning, and I/O time partitioning) can be easily implemented in file/storage servers, network-attached storage appliances, or distributed file/storage systems. Although these other systems may present additional sources of potential unpredictability, we expect disk-driven unpredictability to still dominate. The only requirement in these implementations is that the closest driver to the disks must be able to identify the VMs from which the accesses are coming.

Our evaluation demonstrates that VirtualFence improves I/O predictability (or, equivalently, that it reduces deviation) significantly, as long as we utilize all of its component techniques at

the same time. In fact, we show that simply using an SSD as a cache of HDD data is *not* enough. More fundamentally, our evaluation illustrates the tradeoff between predictability and performance: the more we improve predictability, the worse average response time becomes. The challenge is finding the smallest response time that will produce enough predictability.

Summary of contributions. We study the impact of VMM architecture, approach to storage virtualization, and storage device on I/O predictability. We propose VirtualFence, a system that combines SSDs working as HDD caches, space and non-work-conserving time partitioning. We quantify the impact of each feature of VirtualFence on I/O predictability. Using VirtualFence, we investigate the fundamental tradeoff between predictability and performance.

The remainder of the chapter proceeds as follows. The next section motivates our work. Section 4.2 motivates our work. Section 4.3 describes our experimental methodology and workloads. Section 4.4 details the results from our VMM characterization. Section 4.5 describes VirtualFence, whereas Section 4.6 presents its evaluation. Section 4.7 discusses different aspects of VirtualFence and our results.

4.2 Motivation

Many users desire performance predictability. Although most cloud users may not require predictable VM performance, many actually do. For example, streaming (video/audio) and gaming applications typically seek to achieve a consistent rate (*e.g.*, displayed frame rate) rather than the highest performance, if that performance might introduce unpredictability (jitter). There are also many cases where repeatable behavior is important, such as performance tuning, debugging, and diagnosis [27, 72]. In fact, it is impossible to evaluate the impact of changes to an application in the cloud, if its performance may constantly be affected by consolidation. Finally, many applications implement workflows/pipelines (*e.g.*, Nutch [8], genome analysis [39]), where the performance that can be achieved in each stage depends on the expected performance of a previous stage. Properly designing such applications for the cloud is

impossible if the performance of each stage can vary widely.

Predictability would benefit cloud providers and users. As predictability is important to many users, we argue that IaaS cloud providers should offer a new class of predictable-performance service. Importantly, this class of service should *not* replace their existing (predictability-oblivious) services. *Users who do not require predictability can still use the existing services.* Rather, the new class should be an additional service that uses a separate set of tightly managed hardware resources. (Section 4.7 discusses combining the two classes of service onto the same hardware infrastructure.) The amount of resources to be purchased for the new class of service can be small at first, and be increased as demand for predictable behavior increases. Current IaaS providers already offer a range of other classes of service, such as the Cluster Compute and Cluster GPU service classes of Amazon EC2.

The tight management of resources in this class of service would: (1) enable the provider to charge for exactly the pre-defined levels of performance and predictability that its users require; (2) enable the provider to conserve energy when resources are not used to guarantee the performance paid for by its users. For example, unneeded CPUs or memory modules can simply be turned off. In fact, the tight management of resources allows the provider to provision just enough servers, for additional (operating and capital) cost savings. Obviously, current services can use fewer servers by overbooking resources, but they cannot provide predictability; and (3) create an obvious relationship between the resources that customers pay for and the performance that can be achieved with those resources, i.e. users never complain that the performance of their VMs suddenly got worse (when the provider stopped dedicating more than the minimum set of contracted resources). Gulati *et al.* mention some of these same benefits to limiting maximum allocations [45].

Cloud users can also benefit from predictability for three reasons: (1) they can rely on it to implement applications for which predictability is more important than receiving as many resources as are available; (2) predictability can lower their cloud costs when the provider can

save money by conserving energy or provisioning their data centers more tightly; and (3) they can predict their cloud costs into the future with the certainty that their VMs' performance will never be affected by changes in provider-side resource allocation.

Importantly, our approach enables a wide variety of performance and predictability levels. For example, a user may purchase $1/n$ (n is defined by the provider) of a PM and receive the performance that this fraction of resources produces. If this performance is not good enough, the user can purchase any multiple s ($s \leq n$) of this fraction. The larger the fraction, the more consistent the performance will be. Ideally, the price of a VM instance with s slices of a PM in the predictable service would be the same as (or only slightly higher than) an instance with the same amount of resources on average in the existing predictability-oblivious service.

Client-side throttling does not work. One might think that providing an additional class of predictable service is unnecessary, as delays can be added on the client side to achieve predictable behavior. However, this intuition is incorrect. As the client does not know how bad VM performance may get in the future (the *WorstPerf* performance mentioned above), it cannot target a performance level that is guaranteed to be consistent. In fact, even if the client knew the value of *WorstPerf*, it would have to set its delay to achieve this worst performance. Any other value could be exceeded.

4.3 Methodology

In this section, we first define the metric we use to evaluate VM performance predictability. We then describe the virtualized systems we study in Section 4.4, each of which comprises a different combination of VMM, approach to storage virtualization, and storage device. Finally, we describe the experimental environment, including the workloads and hardware execution platform.

4.3.1 Performance Deviation Metrics

It is difficult to study performance predictability without a metric to quantify it. Since our notion of predictability means achieving the same VM performance in the presence of other VMs as in isolation, we earlier defined the performance deviation metric informally to relate these quantities explicitly. Again, we measure performance deviation as the percentage performance degradation when a VM runs in the presence of other VMs compared to when it runs alone on the physical host. Specifically, let P_I be the (initial) performance of a VM when running alone, and P_D be the (degraded) performance of the VM when co-located with other VMs. Then, the amount of deviation δ is:

$$\delta = |\frac{P_I - P_D}{P_I}| \times 100\%$$

When multiple (say n) VMs executing the same workload are used in an experiment, we report the average deviation:

$$E[\Delta] = \frac{\sum_{i=1}^{n} \delta_i}{n}$$

where $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$.

As we show below, performance deviation is often different for throughput and response time. Thus, throughout the paper, we study deviation for both metrics.

4.3.2 Virtualized Systems

We measure performance deviation across four well-known VMMs and two types of persistent storage, SSD and HDD. First, we study VMWare's Workstation (8.0), where the host OS is responsible for device I/O. In contrast, we also study VMWare's ESXi Server (5.0), where guest VM I/O operations trap into the VMM, which then directly accesses the I/O device. Third, we study Xen (4.0.1), where a split-driver model is used for device I/O. Here, an isolated device domain (Dom0) runs the device drivers. Therefore, VMs (guest VMs, which are referred to as DomU) pass their I/O requests through to Dom0 on I/O accesses. Like VMWare's ESXi, the

Xen VMM runs on the hardware directly. Finally, we study KVM (0.12.5), where the Linux kernel is equipped with native virtualization capabilities. As such, it relies on the Linux kernel to actually accomplish device I/O.

As VMWare Workstation and KVM run on top of Linux, and Xen incorporates drivers from Linux, we also run our workloads as processes on a Linux setup.

Finally, we study both file-based and disk-partition-based storage of VM persistent data in Xen and KVM.

4.3.3 Workloads

We use workloads from Filebench [79], a popular framework for measuring and comparing file system performance. Specifically, we use Fileserver, Mailserver, and Webserver. Fileserver emulates a server hosting directories owned by multiple users; Mailserver focuses on mail operations and has an I/O mix of a read per sync write; and Webserver emulates a server that services a read-only workload.

To measure deviation, we compare a VM's I/O performance when running alone against that when running with three other VMs. Specifically, we configure workloads of four VMs running concurrently, each of which executes the same Filebench application. One of the VMs is configured to produce a low-intensity I/O load that is approximately 8% of the storage system's saturation load. Each remaining VM is configured to produce approximately 24% of the storage system's saturation load. (We produce a load of x% of saturation by finding the saturation throughput for each application, and adjusting the number of threads and thread I/O rate to achieve x% of that throughput.) Overall, the four VMs reach 80% of saturation, representing an aggressive consolidation scenario. We then compare the low-intensity VM's performance to when it runs alone, and the performance of each of the three higher-intensity VM to when it runs alone. Note that we scale the load to maintain a constant utilization level (80%) across storage systems (HDD, SSD, and VirtualFence).

We call the above setup a 4-VM heterogeneous workload and use it as the primary workload for our study for two reasons. First, we want to study how higher-intensity VMs affect the predictability of low-intensity VMs. Second, we want to study deviation when VM consolidation leads to high utilization levels. In Section 4.6, we also study homogeneous workloads and systems with low-intensity VMs only, resulting in low aggregate loads.

4.3.4 Experimental Platform

We run our experiments on a server equipped with a 2.4GHz 4-core Xeon CPU (each core supports two hardware threads), 8GB of RAM, a 60GB SSD, and a 160GB 7200RPM SATA HDD. According to its datasheet, the HDD has an average seek time of 11ms and full stroke time of 22ms. The SSD is spec'ed with random read performance >20,000op/s and random write performance >5,000op/s. We measured erasures, including garbage collection, to take approximately 3.5ms-4ms in a write-only benchmark. The guest OS in the VMs is always a Debian installation with Linux kernel version 2.6.32. The host OS for VMWare Workstation and KVM is the same Linux installation.

The Linux 2.6 kernel has 4 commonly used disk schedulers: Noop, Deadline, Anticipatory, and Completely Fair Queuing (CFQ). We set the disk schedulers of the guest and host (including Xen's Dom0) Linux systems to Noop and CFQ, respectively. We choose Noop in the guest OS to isolate the impact of the VMM's I/O scheduling. We choose CFQ for the host OS because it minimizes deviation when not using VirtualFence.

In all experiments, we allocate 512MB of memory to each VM and pin it to a core to minimize the impact of VMM CPU scheduling. We run at most 4 VMs simultaneously so that each VM can be allocated an entire core.

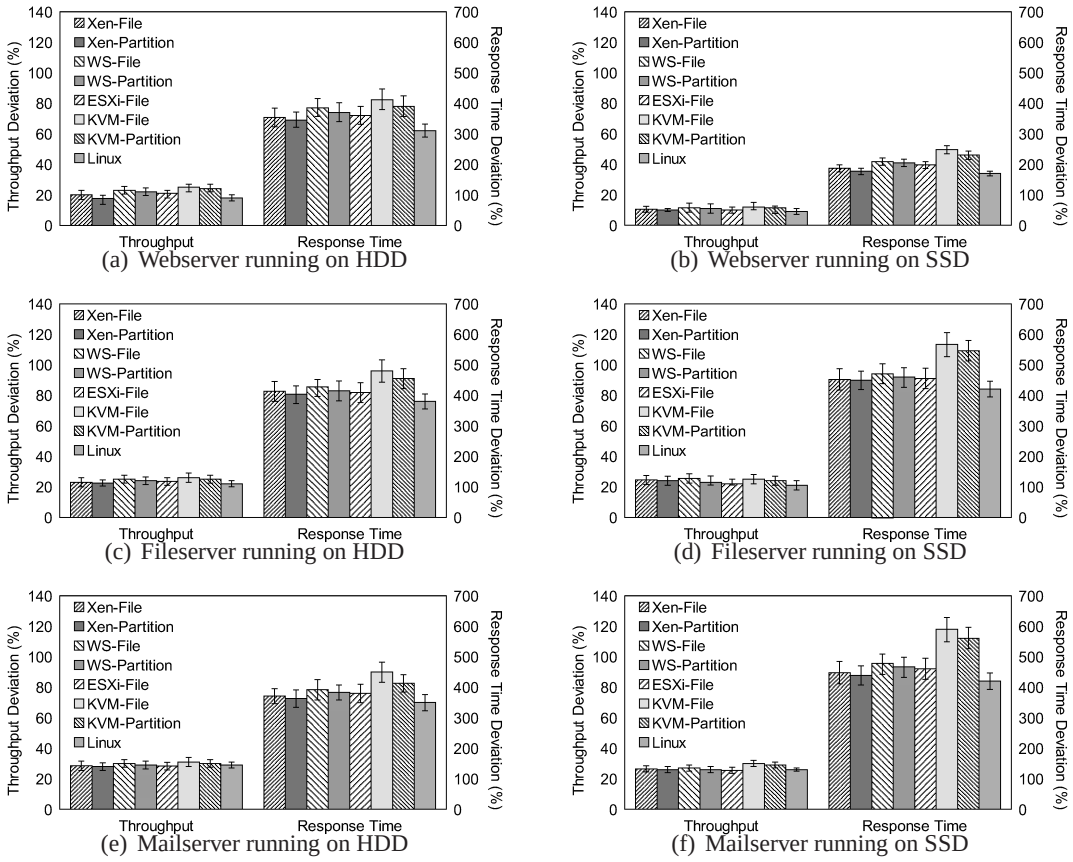


Figure 4.1: Performance deviation experienced by the low-intensity VM in 4-VM heterogeneous workloads.

4.4 VMM-Driven Performance Deviation

Our study begins with characterizing I/O performance deviation for the four different VMMs, when using file-based vs. disk-partition-based storage virtualization, and when using an SSD vs. an HDD device. Figure 4.1 plots the deviation experienced by the low-intensity VM in the 4-VM workloads described in Section 4.3.3. We plot deviation for both throughput and response time. In both cases, the lower the measure, the better. The range markers represent the minimum and maximum values from three experiments, whereas the bar represents the average. We do not show the results for the high-intensity VMs because they exhibit similar trends.

Many interesting observations arise from these graphs. First, deviation is endemic across

all system configurations for both throughput and response time. For example, Xen's deviation ranges from ~17-32% for throughput and ~327-433% for response time when running on an HDD, and ranges from ~9-28% for throughput and ~166-479% for response time when running on an SSD.

These high deviations show that performance degrades significantly when running with other VMs. Much of the degradation arises from interference between the 4 VMs. The higher aggregate workload leads to increased queueing times, resulting in higher response times. When running on the HDD, interleaved requests from multiple VMs increase seek overheads, resulting in loss of throughput and higher response times. Interestingly, the response time deviations for workloads with writes (Fileserver and Mailserver) are worse when running on the SSD compared to the HDD. The main reason is that SSD writes sometimes cause expensive Flash block erasures, which may affect accesses from all VMs.

Second, deviations for all workloads and HDD/SSD combinations are worse when a VMM is used vs. stand-alone Linux. Since many VMMs operate on a Linux base, such as KVM or Xen's Dom0, this further demonstrates that VMMs intrinsically increase deviation.

Third, file-based virtual disks exhibit slightly worse predictability than partition-based virtual disks. Two potential sources for the greater deviation are the file system code running inside the VMMs, and the need to update the metadata of the files implementing virtual disks when the hosted VMs access their virtual disks.

Finally, Figure 4.1 shows that KVM exhibits the highest deviations. The main reason is that, by default, KVM propagates writes coming from the VMs directly through to the storage device to improve reliability. When this feature is disabled, the KVM deviations become comparable to those of the other VMMs.

Although the results are not shown here, we have also measured performance deviation for a low load scenario, where the aggregate load reaches only 20% of saturation. Throughput deviation is relatively low in this scenario but response time deviation is still significant.

Taken together, these observations mean that current systems lack I/O predictability. Large deviations may arise from the resource allocation policies; specifically, work-conserving policies link VMs' I/O resource allocation to the number of co-located VMs, causing deviations as this number changes. Large deviations may also arise from device-specific characteristics. Interleaving HDD requests from different VMs leads to higher seek overheads, whereas SSD erasures initiated by a VM can interfere with operations from other VMs.

4.5 VirtualFence

VirtualFence uses three techniques to reduce performance deviation between VMs whose virtual disks are stored on the same physical disk: (1) a non-work-conserving time-division I/O scheduling algorithm with relatively coarse-grained time quanta, (2) a small persistent SSD cache in front of a much larger HDD, and (3) space partitioning of both the HDD and the SSD cache.

The non-work-conserving time-division I/O scheduling serves two purposes. First, it ensures that the resources allocated to a VM are (mostly) constant regardless of the number of co-located VMs. Second, it avoids fine-grained interleaving of requests from different VMs to reduce inter-VM interference; for an HDD, this reduces seek overheads between operations from the same VM, whereas for an SSD, it reduces the interference of erasures from one VM on accesses from other VMs.

Despite the non-work-conserving policy, a system with only HDDs would still suffer some performance deviation when multiple VMs are co-located; as the system switches from serving 1 VM to another, the HDD's head will have to move across partitions, leading to higher seek time for the first HDD operation, and so performance deviation. We limit the impact of this deviation by putting the SSD cache in front of the HDD. With a reasonable hit ratio in the SSD cache, we may eliminate some of these expensive HDD accesses. Moreover, the SSD cache significantly increases the performance of the virtual disk, so that the expensive first HDD

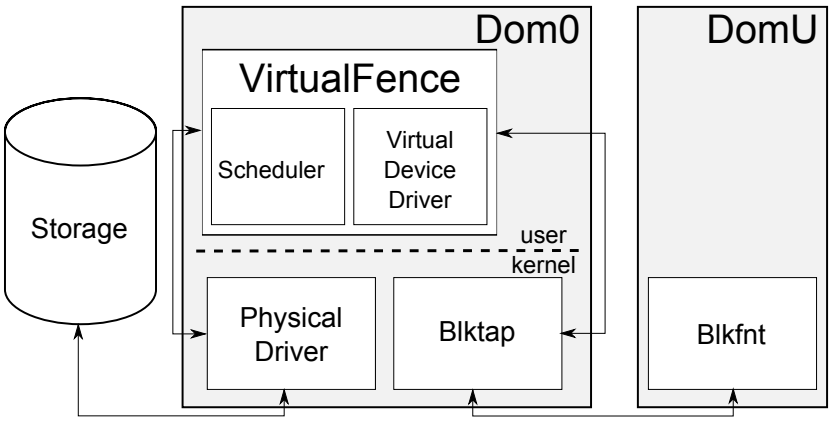


Figure 4.2: VirtualFence architecture.

operation is amortized across many more operations.

Finally, the space partitioning of the HDD limits the seek overheads between operations from the same VM, while the space partitioning of the SSD cache is a non-work-conserving space allocation scheme to ensure constant cache allocation for each VM.

Note that VirtualFence deals solely with storage I/O resources, assuming that other resources (e.g., CPU cores and memory) are also managed with predictability-preserving schedulers.

4.5.1 Prototype

We have implemented a prototype VirtualFence system in the Xen VMM version 4.0.1, using the blktap user-level toolkit [82]. This prototype includes a device driver and a scheduler. The device driver instances—a separate instance of the device driver is used to service each distinct virtual disk—and the scheduler each runs as a user-level process in Dom0 (Figure 4.2). We have not experimented with multiple SSDs and HDDs but it should be trivial to extend our prototype, as long as the caches for virtual disks co-located on a single HDD are themselves co-located on a single SSD. We discuss multiple SSDs and HDDs again in Section 7.

The SSD cache holds two types of persistent data: (1) blocks cached from the HDD, and (2) metadata describing the state of each cache block (e.g., valid bit, HDD block number). The

driver implements the data structures needed to support an LRU replacement policy in volatile memory, including an LRU list of blocks, a write list that points to dirty blocks that need to be written to the HDD and then evicted, and a free list. At start up, the driver scans the SSD for all metadata, and builds all the in-memory data structures. No “last usage times” are kept across system restarts.

The LRU maintenance is simple. A background thread attempts to maintain the size of the free list above a threshold size by evicting the oldest entries in the LRU list as needed. Dirty blocks to be evicted are moved to the write list while the writes to the HDD are outstanding. If the free list ever reaches a low watermark threshold, processing of incoming requests is halted until the free list grows above the low watermark.

The driver uses asynchronous I/O to read and write data from/to both the SSD and HDD.

4.5.2 Space Partitioning

VirtualFence uses a separate partition of an SSD as a cache for each virtual disk co-located on the same HDD. (Note that while the sizes of the partitions in our evaluation experiments are the same, it is trivial to make the cache size proportional to the size of the virtual disk, so that larger virtual disks also have larger caches.) We have also implemented a variation that uses a single SSD partition as a shared cache across multiple virtual disks to quantify the impact of space partitioning on deviation.

The implementations of the two variants are slightly different. In the space-partitioned version (i.e., VirtualFence), the caching code runs inside the driver process that manages each virtual disk. In the shared-cache variant, the caching code runs in a separate process (that must then interact with the multiple drivers managing the virtual disks sharing the cache). This structure makes the shared cache implementation slightly less efficient than VirtualFence because of the inter-process communication between the cache manager and the drivers.

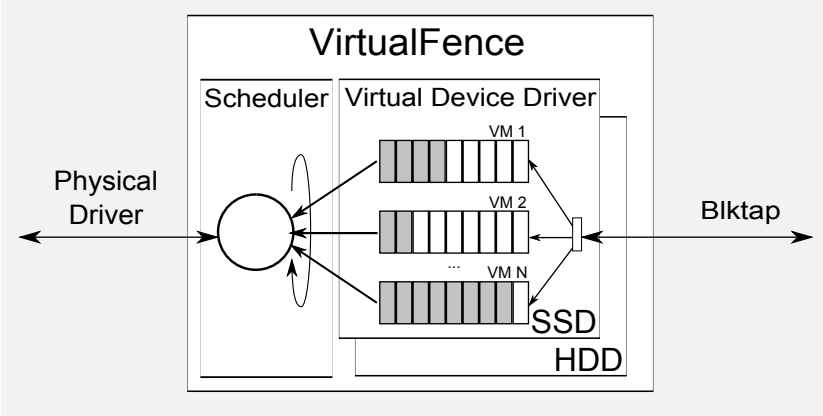


Figure 4.3: Driver with non-work-conserving time partitioning.

4.5.3 Time Partitioning

Our non-work-conserving I/O scheduler assumes that a physical SSD/HDD pair is used to service at most *n* simultaneous active virtual disks, and so divides access to the physical disks into *n* equal-sized time slots. When a VM starts running on a host, its virtual disk is allocated one or more I/O slots (depending on how much of the host's I/O resources is assigned to that VM). The scheduler then round-robins between the slots, *leaving a slot idle* when it is unassigned or the assigned virtual disk does not have any I/O activities; utilizing these slots would break the non-work-conserving property of the scheduler. On the other hand, this property of the scheduler also impacts performance, as we discuss extensively in Section 4.6.3. Figure 4.3 illustrates our implementation.

The driver translates each user I/O request into requests to the SSD and HDD, and adds each type of requests to the appropriate device I/O queue. Each virtual disk has a distinct set of queues that are serviced during the slots assigned to the VM.[1] The scheduler informs a driver instance when its assigned slot is scheduled, at which time it is allowed to forward requests to the SSD and/or the HDD until the slot time expires. A driver can end its slot early (see below), in which case the scheduler will lengthen the slot time appropriately the next time that slot is

[1] We currently assume that each virtual disk is only attached to a single VM and that a VM attaches to at most one virtual disk.

scheduled. If a driver overruns the slot time, the scheduler will deduct the overrun from the next slot.

To implement accurate time partitioning without losing performance, we need to send as many accesses as possible in a slot without running over the time allocated to the slot. In addition, it is more efficient to batch requests because of effects such as disk scheduling and fixed access overheads. Thus, our approach is to estimate the service times of batches of accesses, and to send the largest batch of accesses that is estimated to complete within the remaining time in the slot.

Our driver dispatches requests to the SSD and HDD in the same manner as follows. If there are no pending requests, then wait until a request arrives or the current slot terminates. If there are pending accesses, and at least the first access is estimated to complete within the remaining time in the slot, find the largest batch that is estimated to fit within the remaining time. (We explain our prediction model below.) After the completion of a batch of requests, if time remains in the slot, then repeat.

The driver will end a slot early if the first pending HDD request is estimated to take longer than the remaining time in the slot. This is because HDD resources are much more constrained than the SSD, and thus, when the remaining slot time cannot be used for accessing the HDD, it is better to "credit" it to the next slot instead of wasting it. On the other hand, if a batch of HDD requests overruns the slot time, while waiting for the batch to complete, slowly send SSD requests to not waste this time while not causing even more slot delay by having to wait for the completion of a large batch of SSD requests.

Predicting HDD request service times accurately can be quite complicated [94, 108]. For our purposes, however, it is sufficient to use a simple piece-wise linear function that predicts the access time of a request based on the distance between the block being requested and the block requested by the immediately preceding request. When predicting the service time of a batch, we order the requests using the block addresses under the assumption that the disk

scheduling algorithm includes some form of scanning. We parameterize the prediction function for our specific HDD by benchmarking the service time of a large number of random batches of accesses, each batch with a random mix of reads and writes. Our approach leads to reasonably accurate prediction of batch service time: for a benchmark issuing batches of random sizes averaging 50 accesses and 336ms batch service time, over 70% of our predictions are within [−5ms,5ms] of the actual batch service times.

Measurements of the SSD used in our experiments show that request service times can be approximated using a linear function that depends on the number of requests simultaneously submitted to the asynchronous I/O system. We parameterize the prediction function for our SSD by benchmarking the service times of a large number of batches of accesses, where each batch has a random size (between 1 and 100), a random split between reads and writes, and random target blocks. For a benchmark with an average batch size of 49 accesses and an average batch service time of 19.4ms, over 83% of our predictions were within [−250us,250us] of the actual batch service times.

4.6 Evaluation

We now explore VirtualFence's effectiveness in providing performance predictability. All experiments are performed using the workloads and experimental platform described in Section 4.3. The SSD cache block size is set to 4KB to match the default 4KB block size of the HDD. We also adjust the SSD cache size to explore the impact of different hit rates. We use the notation VirtualFence(X%,Yms) to denote a VirtualFence system with a time-sharing slot size of Yms, and the SSD cache empirically sized to achieve a hit rate of X%. We explicitly set the SSD cache hit rate to systematically isolate its impact; in practice, administrators would set the SSD partition size (and the number of time slots) for each VirtualFence virtual disk based on the QoS/resources promised to the disk's owner and the number of virtual disks to be consolidated on the physical server.

Variant	HDD	SSD	Cache	NWC
HDD+NWC	x			x
SSD+NWC		x		x
Hybrid/Shared	x	x	Share	
Hybrid/Shared+NWC	x	x	Share	x
Hybrid/Partitioned	x	x	Partition	
VirtualFence	x	x	Partition	x

Table 4.1: Variants of VirtualFence comprising different combinations of predictability-enhancing techniques. The HDD and SSD columns show whether a variant uses an HDD and SSD device, respectively. When both devices are used, the SSD acts as a cache for the HDD. The Cache column shows whether the SSD cache is shared or partitioned. The NWC column shows whether non-work-conserving time partitioning is used.

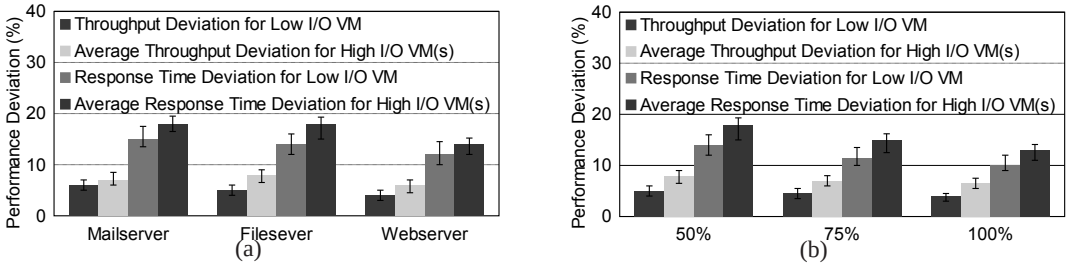


Figure 4.4: Deviation when running (a) the 4-VM heterogeneous workloads on VirtualFence (50%,20ms), and (b) the 4-VM heterogeneous Fileserver workload on VirtualFence (X%,20ms), with $X \in \{50\%, 75\%, 100\%\}$. The range markers show the minimum and maximum values from three experiments whereas the bar shows the average.

To isolate the contributions of the different features of VirtualFence toward increasing predictability, we also measure deviation for many incomplete variants of VirtualFence. Table 4.1 lists these variants. The first two variants, HDD+NWC and SSD+NWC, are designed to isolate the benefits of non-work-conserving time partitioning. The Hybrid/Shared variant uses an SSD cache in front of the HDD, but the entire cache space is shared between multiple virtual disks. Hybrid/Shared+NWC extends this variant with non-work-conserving time partitioning. Hybrid/Partitioned is VirtualFence without non-work-conserving time partitioning, isolating the benefits of space-partitioned SSD caches.

4.6.1 Performance Deviation

VirtualFence. We begin by showing VirtualFence's effectiveness at reducing performance deviation. Figure 4.4(a) shows the measured deviation when running the 4-VM heterogeneous workloads on VirtualFence(50%,20ms). Figure 4.4(b) shows the measured deviation when

running the 4-VM heterogeneous Fileserver workload, which experiences the highest deviation, on VirtualFence with hit rates ranging from 50% to 100% and a time-sharing slot size of 20ms.

Figure 4.4(a) shows that VirtualFence is successful at reducing deviation in both throughput and response time, compared to a system without VirtualFence (Figure 4.1). In fact, VirtualFence produces lower deviations regardless of storage device or approach to virtualizing storage. For the low-intensity VM, all deviations are ≤15%, compared to throughput deviations of ≥31% and response time deviations of ≥443% without VirtualFence. Furthermore, deviations are always lower than 19% when the SSD cache affords a 50% hit rate. Figure 4.4(b) shows that deviation decreases as the SSD hit rate increases.

The results are positive in terms of raw performance as well. For example, Fileserver file accesses (97KB on average) by the low-intensity I/O VM take an average of 19ms, when the VM runs in isolation on the HDD configuration. When the same VM runs co-located with 3 high-intensity VMs, the average file access time increases to 98ms. We increase the I/O intensity of each VM by a factor of 3.3x in VirtualFence(50%,20ms) to achieve the same utilization as in the HDD case. Despite the much higher I/O intensity, the low-intensity I/O VM experiences an average file access time of 59ms when running alone, and just 64ms when co-located with 3 high-intensity I/O VMs, under VirtualFence(50%,20ms).

Isolating the contributions of different features. Figure 4.5 plots performance deviation when the Mailserver workload is run on the variants (including the full VirtualFence implementation) listed in Table 4.1. Performance deviations for HDD and SSD (from Figure 4.1) are also shown as baselines. These results are representative of all three Filebench workloads.

First, this figure shows that VirtualFence achieves performance predictability close to that of SSD+NWC. Specifically, SSD+NWC achieves 3% and 10% throughput and response time deviation, respectively, whereas VirtualFence achieves 6% and 12%. SSD+NWC represents the best case scenario since it includes space partitioning (each VM is given a separate SSD partition), non-work-conserving scheduling, and storage completely on the SSD. The fact that

these two systems achieve almost the same predictability shows that our caching approach is effective, allowing VirtualFence to extend the predictability benefits of (expensive) SSDs to much larger (and cheaper per byte) HDDs with small SSD caches.

Second, results for HDD+NWC suggest that non-work-conserving time partitioning can also be effective in reducing deviation when not using an SSD cache. However, the movement of the disk head between partitions when changing between time slots assigned to different VMs is sufficiently large that HDD+NWC with a 20ms slot size still incurs a 13% throughput deviation and a 33% response time deviation. As we show in Section 4.6.3.2, increasing the slot size to attack this source of deviation also increases the response time observed by a VM running alone on a host. As already mentioned, this source of deviation also exists in VirtualFence but is mitigated by the SSD cache.

Third, at this hit rate, non-work-conserving time partitioning achieves higher predictability than using an SSD cache: HDD+NWC has lower deviations than both Hybrid/Shared and Hybrid/Partitioned. Interestingly, HDD+NWC is also better than Hybrid/Shared+NWC, implying that the interference at the shared cache negates some of the benefits of NWC. Of course, as the hit rate increases, the relative advantage of using NWC vs. an SSD cache will likely change.

Fourth, as expected, a shared SSD cache produces worse predictability than a partitioned cache. A shared cache can produce higher absolute performance; e.g., it may benefit an I/O-intensive VM running by itself. However, it would hurt predictability when the VM is co-located with other VMs and so must share the cache.

Finally, all three techniques used in VirtualFence contribute to increasing predictability; VirtualFence achieves higher predictability than the other configurations, except for the much more expensive SSD+NWC.

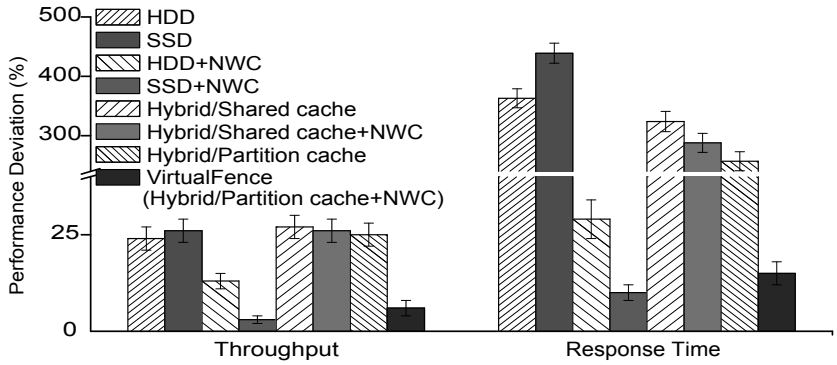


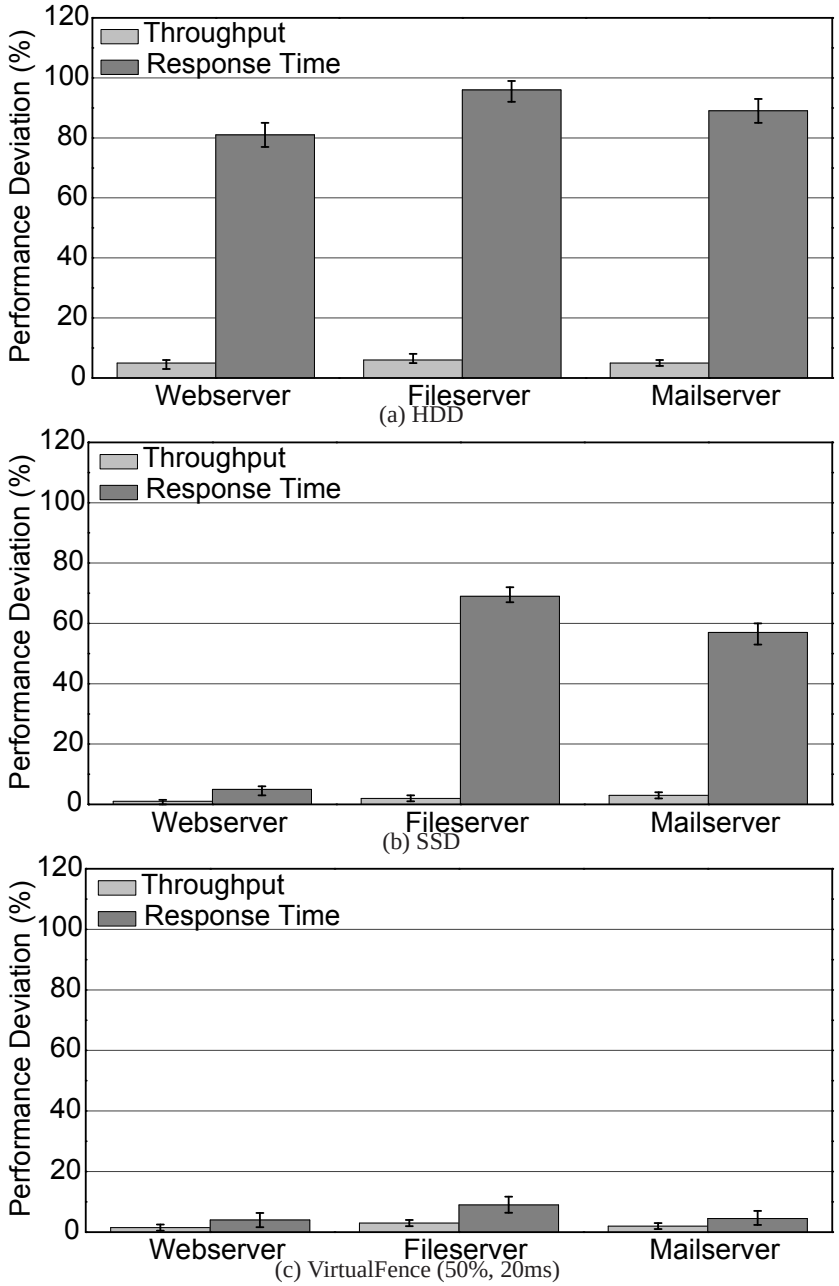
Figure 4.5: Deviation when running on VirtualFence(50%,20ms) compared to various incomplete variants of it. Each bar shows results for the low-intensity VM in the 4-VM heterogeneous MailServer workload. The cache size for Shared-cache versions is equal to the sum of the caches in the Partitioned-cache cases.

4.6.2 Performance Deviation at Low Load

The previous subsection shows that VirtualFence is effective in aggressive consolidation scenarios. In this section, we consider what happens when the aggregate load is low, representing more conservative scenarios.

Figure 4.6 shows the average throughput and response time deviation under low aggregate loads for SSD, HDD, and VirtualFence(50%,20ms). Each workload runs 4 homogeneous VMs, where each VM is configured to generate 5% of the storage system's saturation load (i.e., each VM is less I/O intensive than any VM we have discussed so far). These results show that, even at these low loads, HDD experiences very high deviation. This is because requests from multiple VMs are interleaved, leading to much higher seek overheads. Although throughput deviation for the SSD is low, response time deviation is still significant for the workloads with writes (greater than 50% for both Mailserver and Fileserver). All deviations are below 14% for VirtualFence.

The raw performance results for this low-load workload are interesting as well. For example, Fileserver file accesses by each VM take an average of 21ms, when it runs in isolation on the HDD configuration. When the 4 low-intensity I/O VMs are co-located, the average file access time increases to 42ms. We increase the I/O intensity of each VM by a factor



(a) HDD

(b) SSD

(c) VirtualFence (50%, 20ms)

Figure 4.6: Deviation when running 4-VM homogeneous, low-rate I/O workloads.

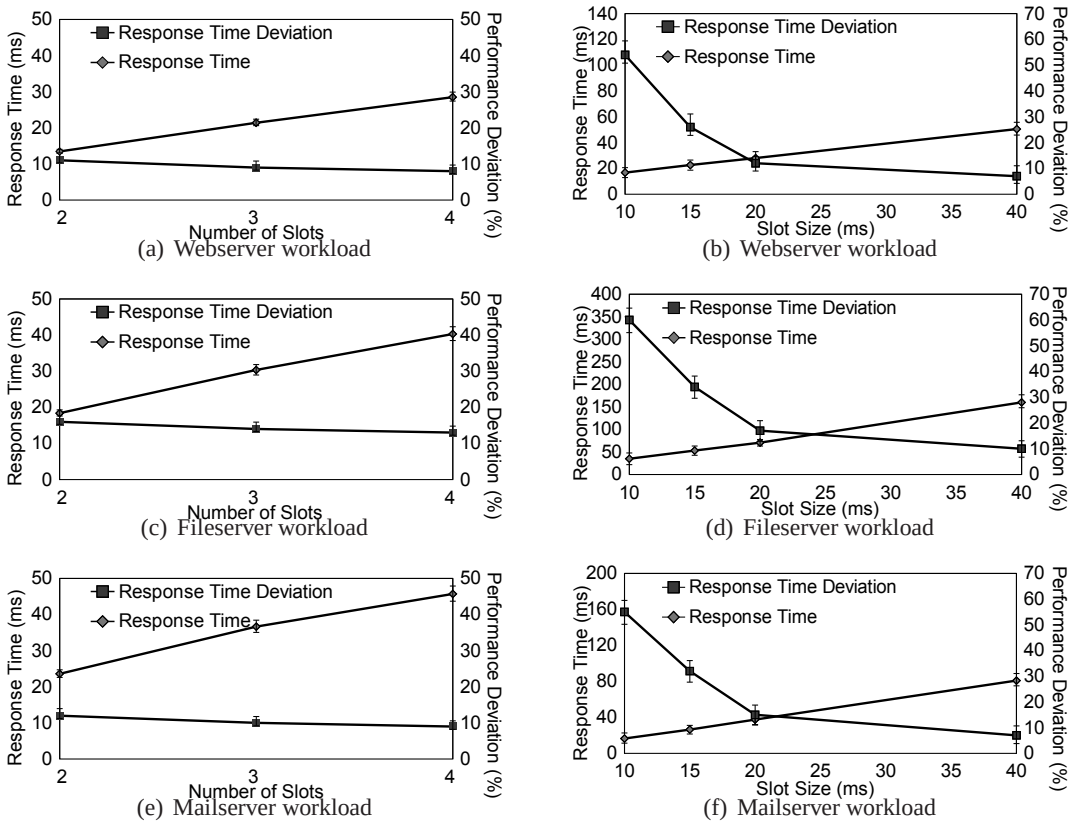


Figure 4.7: Number of slots and response time trade-off. Slot length and response time trade-off.

of 2.7x in VirtualFence(50%,20ms) to achieve the same utilization as in the HDD case. Despite the much higher I/O intensity, each VM experiences an average file access time of 35ms when running alone, and just 40ms when it is co-located with the other 3 VMs, under VirtualFence(50%,20ms).

4.6.3 Performance vs. Predictability

In this subsection, we explore the performance vs. predictability tradeoff that VirtualFence exposes. In particular, we explore the performance deviation and raw performance impact of its two key parameters: the number of VM slots per PM, and the length of each slot.

4.6.3.1 Impact of Number of Slots

The number of VM slots determines how aggressively a cloud provider will be able to consolidate VMs onto the same PMs. Figure 4.7 illustrates the impact of the number of slots on response time deviation and raw response time of VirtualFence(50%,20ms). For each number of slots *n*, we assume *n* VMs, and configure each VM to generate only ~5% of the VirtualFence(50%,20ms) saturation load. Such a low aggregate load is a particularly challenging for VirtualFence raw performance-wise, because it may produce significant average waiting times.

As the figure illustrates, VirtualFence produces lower response times as we decrease the number of slots (while keeping the slot length fixed). The reason is that fewer slots also means lower waiting times, as each VM is allotted a higher fraction of time. From the opposite point of view, raw response time worsens linearly with increasing number of slots, because VirtualFence will not service a request until its corresponding slot is scheduled.

Interestingly, response time deviation decreases slowly as we increase the number of slots. With a large number of slots, deviation would approach 0%, because the waiting time would overwhelm the single disk head movement in the first request of each slot.

Clearly, there is a tension between wanting a small number of slots to reduce average response times and wanting to increase the number of slots to improve predictability. Fortunately, VirtualFence makes predictability reasonably good even with only a few slots. Thus, we would like to set the number of slots at the smallest number that will enable enough consolidation.

4.6.3.2 Impact of Slot Length

The key source of remaining deviation in VirtualFence is the need to move the disk head from one partition to another when changing slots assigned to different VMs. Thus, the slot length directly impacts VirtualFence's predictability: a longer slot better amortizes the inter-partition head movement cost among more requests. However, lengthening the slots also increases response time, because I/O operations issued outside of a VM's slot incur greater delay.

Assuming 4 slots, Figure 4.7 plots the response time deviation and average response time, as a function of slot length for VirtualFence(50%,10-40ms). We use the 4-VM heterogeneous workload (Section 4.3.3), and focus on the low-intensity I/O VM in Figure 4.7. This setup is challenging for VirtualFence predictability-wise, because it is almost certain that every first access in the low-intensity VM's slot will cause a disk head movement.

The figure clearly shows the tradeoff between lowering deviation by lengthening the slots against increased response time. For all workloads, lengthening the slots from 10ms to 20ms significantly reduces performance deviation. Further lengthening the slots to 40ms reduces deviation much more slowly at the expense of a further, essentially linear, increase in response time. Thus, a slot length of 20ms is the right tradeoff for our particular SSD and HDD devices. We have chosen this length as our a default for all previous experiments based on these results (and additional ones not shown here).

Again, there is a tension between wanting shorter slots for lower average response times and longer slots for better predictability. The slot length should be the shortest that will produce enough predictability.

4.7 Discussion

Recall from the Introduction that the goal of VirtualFence is to produce consistent raw performance between $WorstPerf$ and $BestPerf$, the extremes in performance in a predictability-oblivious, work-conserving scenario. However, to achieve this goal, VirtualFence must be properly configured as demonstrated in the previous section. Determining the best configuration involves experimenting with the devices at hand, and understanding how much users value performance vs. predictability. Since we propose VirtualFence for a new class of predictability-conscious cloud service, we expect our users to accept relatively low (but consistent) performance in exchange for good predictability. This would mean a tendency to prefer longer slots. Given that predictability is good even with few slots, the cloud provider can choose to use

more slots (as long as performance is still acceptable to users) to enable more aggressive consolidation (lower costs). *As a target for "acceptable performance", the cloud provider can estimate $WorstPerf$ by using its existing (predictability-oblivious) service* and desired amount of consolidation. This value can be used in limiting the number and length of slots.

Clearly, in the predictability-conscious service, resources may go underutilized. The provider may then be tempted to adjust the VirtualFence parameters dynamically to adapt to current workloads and their I/O activities. Unfortunately, doing so could ruin predictability. *A better approach may be to combine the predictability-conscious and predictability-oblivious services onto the same hardware infrastructure.* For example, a VM that requires predictability could be given a fixed fraction of a PM (e.g., a slot of 20ms out of every 100ms, one-fifth of the SSD cache, and a separate disk partition), whereas many co-located predictability-oblivious VMs could fight for the remaining resources.

Importantly, VirtualFence enables cloud users to pay only for the performance that they (consistently) get. If they desire better performance, *they can purchase multiple slots while still retaining predictability.* Given that our system enables the cloud provider to reduce its costs through better resource provisioning and energy conservation, the user may end up paying roughly the same for multiple slots as she would pay for the equivalent of one slot in the absence of VirtualFence.

Finally, it is important to discuss two aspects of our study. First, VirtualFence does not partition *all* I/O resources across VMs. In particular, it does not partition the buffer cache in Xen's Dom0. We made this decision because (1) we find the hit ratio in that cache to be very low; and (2) partitioning the SSD space and the I/O access time is substantially more important for predictability. The low performance deviations that VirtualFence is able to achieve justify our choice.

Second, our evaluation of VirtualFence focuses on a single HDD (and associated SSD). However, our approach extrapolates to RAID or JBOD systems using a single SSD for caching.

The reason is that VirtualFence would be built into the driver closest to the disk array and, thus, could partition the SSD space and I/O access time like the array were a single disk.

VirtualFence can be extended to proportional share scheduling [64], which requires users to translate workload priority into number of slots. VirtualFence can also be extended to support max-min fairness resource allocation [16]. Since max-min fairness is achieved if and only if an increase of any rate within the domain of feasible allocations must be at the cost of a decrease of some already smaller rate, an offline pre-allocation of resources is needed to achieve max-min fairness in VirtualFence.

4.8 Summary

In this chapter, we quantified the impact of storage medium, and VMM architecture and configuration on I/O performance predictability. The results showed that unpredictability is pervasive. Based on these results, we proposed VirtualFence, a software/hardware approach for achieving predictability at low cost. VirtualFence combines a small SSD cache in front of a much larger HDD, and non-work-conserving space and time partitioning. Our evaluation showed that VirtualFence can provide high predictability, as long as all of its features are used at the same time. We also identified and quantified the tradeoff between predictability and performance.

We conclude that it is possible to build performance-predictable storage systems with relatively simple software and hardware components, especially for those users that find predictability just as important as (or even more so than) raw performance.

Chapter 5

Multi-Point Performance Engineering in Server Systems

5.1 Introduction

In this chapter, we address the problem on how to achieve multiple performance targets. Modern server systems encompass multiple components and/or layers containing configuration parameters that can affect performance. Examples include parameters controlling the amount of parallelism (e.g., number of threads), the size and replacement policy used for memory caches, and the scheduling policies for processing workloads. As the complexity of server systems continues to increase, managing the interplay between these configuration parameters to precisely tune performance becomes a challenging task.

This challenge is exacerbated by the need of many service providers to meet multiple performance objectives. For example, reducing the tail latencies of on-line services has received much attention (e.g., [31, 47, 116]). However, techniques and configuration parameter values for reducing tail latencies can often negatively impact performance at other percentiles in the performance cumulative distribution function (CDF). Figure 5.1 shows an example of one such tradeoff in a Web server, where setting the configuration parameter to a small value (i.e., value = 0.1 leading to the purple CDF) can significantly reduce the tail but gives much worse performance for a large part of the space (difference in the purple and blue CDFs between the 15^{th} to 60^{th} percentiles). Thus, administrators must often consider multiple points on the performance CDF when configuring server systems.

Given the challenge of tuning system performance to meet a single performance objective

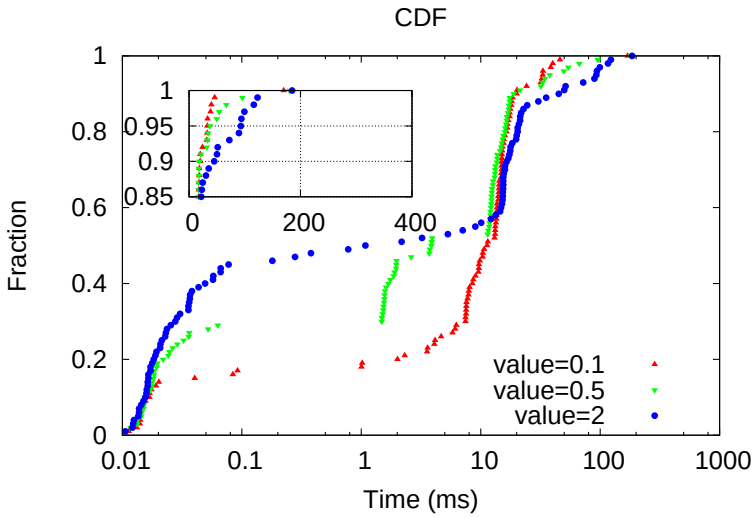


Figure 5.1: Impact of a cache configuration parameter on the response time of a Web server.

(e.g., minimize median response time), it is not surprising that tuning for multiple performance targets is a challenging, error-prone, and time-consuming exercise for server system administrators. In this chapter, we propose OpTune, a framework for guiding administrators to configure server systems to meet specified performance objectives. Examples of performance objectives that can be specified in OpTune include: (1) minimize the average response time; (2) minimize the average response time while keeping the 99^{th} percentile response time below a target value; (3) minimize the 99^{th} percentile response time while keeping the median response time below a target value; and (4) find the "closest" achievable performance CDF for a specified target CDF.

OpTune assumes that administrators can identify a subset of important parameters, which it then carefully calibrates to best achieve the performance objectives. OpTune relies on a graphical representation of the system, performance instrumentation and profiling, and manipulation of performance CDFs to perform its function. The graphical representation describes the main system components and their interactions, and how they aggregate to determine overall system performance. OpTune collects transition probabilities and performance CDFs of the software components at different possible parameter settings during a profiling phase. OpTune relies on the fact that the impact of different parameters are typically localized (e.g., a parameter might

mainly affect the performance of just one component) and independent to reduce the amount of profiling. OpTune then uses the profiled data, the graphical representation, and a small set of mathematical operations on the components' performance CDFs to predict system performance for different sets of configuration parameter values.

Finally, OpTune formulates the configuration problem as an optimization problem, and calculates the percentage of time that the server system should be configured with a particular set of parameter values to best meet the performance objectives. The solution to this optimization problem may be a set of static configuration parameter values. More interestingly, the solution may be several sets of values, each of which should be used periodically (e.g., for 2 minutes every 10 minutes) to achieve a performance CDF over time that is impossible to get with just a single static set of parameter values. Such dynamic configurations would be extremely difficult for administrators to determine manually.

We demonstrate the broad utility of OpTune by prototyping it for a diverse set of widely-used systems. Specifically, we integrate OpTune into a Web server, a filesystem emulator, and the scheduler of a Hadoop MapReduce processing system. In each case, we first study the performance tradeoff involved in the setting of several critical configuration parameters (e.g., caching and scheduling parameters). We then demonstrate that OpTune is able to achieve desired performance profiles through its optimization strategy with low overheads. For example, consider the performance objectives of minimizing the 99^{th} percentile while maintaining a median response time of less than equal to 10ms in a Web server with two important configuration parameters. The best static configuration would yield a median response time of 3.85ms and a 99^{th} percentile time of 96ms. In comparison, OpTune found three sets of parameter values leading to a median response time of 9.83ms and a 99^{th} percentile time of 89.2ms.

Summary of contribution. We proposing and develop the OpTune framework for guiding administrators when configuring server systems to meet a set of performance objectives; We implement OpTune in three diverse server systems to demonstrate its wide applicability; and

we present results from a large set of case studies to show how OpTune can ease the task of performance tuning, particularly when this process involves tradeoffs between multiple points on the performance CDF (e.g., average and/or median vs. tail latencies).

The remainder of the paper proceeds as follows. Section 2.3 describes related work. Section 5.2 describes the methodology and framework of OpTune. Section 5.3 describes our effort in building OpTune into three systems, and Section 5.4 presents our evaluation results.

5.2 OpTune Methodology

5.2.1 Overview

OpTune represents and tunes performance using the entire performance CDF of the system. To use OpTune, the designers of a system must build a service graph representation, where each node in the graph corresponds to the sequential execution of some code, and each directed edge represents control flow. The graph must contain a root node (where the computation starts) and one or more end nodes (where the computation ends). Each directed edge in the graph is labeled with the probability with which execution will pass from the source node to the destination node. Thus, each execution path from the root to an end node represents a potentially different performance behavior, which happens with different probability. Figure 5.2 shows the service graph for a Web server that serves only static content.

OpTune then works as follows (illustrated in Figure 5.3). OpTune begins with running the server system through a warm up phase so that subsequent profiling of the system will accurately represent steady state behaviors. After warm up, OpTune will enter a profiling phase (which should take place under a real or realistic workload). In this phase, OpTune gathers performance data (e.g., request service times) and transition probabilities for the components and edges in the service graph, respectively, as it sets the configuration parameters to different values within their defined domains. Administrators are expected to specify the configuration

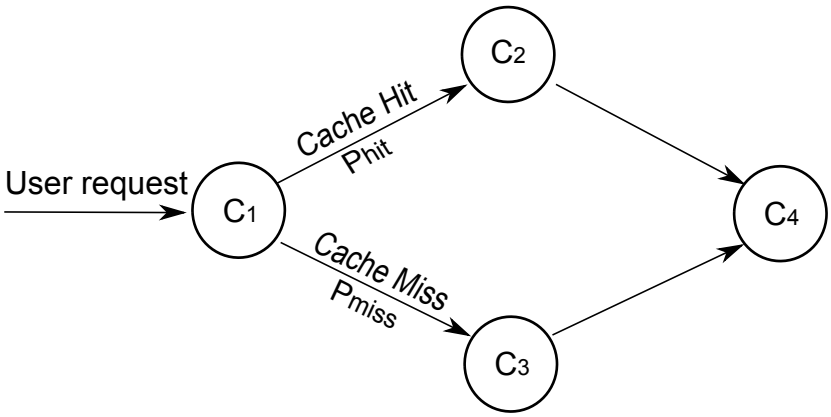


Figure 5.2: The service graph of a Web server. C_1: process user request, C_2: get requested object from the cache, C_3: read the requested object from the filesystem, C_4: compose and send reply to user.

parameters and the values that OpTune should explore. We assume that a server system making use of OpTune is modified to include the necessary calls into OpTune for this profiling.

Once OpTune has gathered the necessary profiling data, it can be directed to enter the configuration phase. In this phase, OpTune sets up an optimization problem to find configuration settings that will best meet a set of performance objectives. OpTune assumes that time will be divided into discrete accounting periods (e.g., 10 minutes), and that its goal is to configure the system to best meet the performance objectives within each accounting period. OpTune then further divides each accounting period into multiple equal length epochs (e.g., 2 minutes) and solves for a set of configurations, one per epoch, that together gives the optimal solution. This sub-division allows OpTune to achieve performance CDFs for an accounting period that would be impossible with a static configuration that should be used throughout the period.

To solve the optimization problem, OpTune must be able to predict the server's performance for different configuration settings. It does this by using the service graph to compose the profiled performance of components and transition probabilities at specific configuration settings. As an example, suppose the Web server whose service graph is shown in Figure 5.2 has two parameters, V_c and V_I. Further, suppose that V_c only impacts C_2 and V_I only impacts C_3. In this case, OpTune would profiles C_2's performance across different settings of V_c (at a

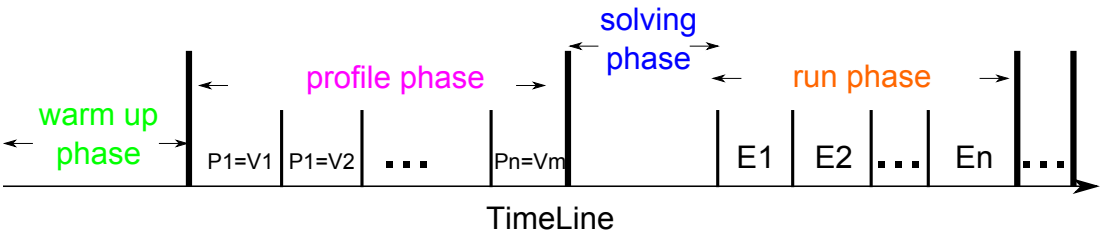


Figure 5.3: An illustration of periodic activities in OpTune.

default setting of V_I), and C_3's performance across different settings of V_I. Then, to predict the server's performance for a particular configuration $V_c = x, V_I = y$, OpTune would compose C_1's performance CDF, C_2's CDF for $V_c = x$, C_3's CDF for $V_I = y$, and C_4's CDF.

Finally, OpTune enters the run phase. If only one configuration setting was chosen, then OpTune configures the system just once. If multiple configuration settings were chosen, then OpTune keeps track of time epochs, and reconfigures the system as appropriate at the beginning of each epoch.

Over time, system performance may deviate from the expected behavior. For example, this can happen when the characteristics of the workload changes. Thus, OpTune continuously monitors performance and compares the observed and expected performance for each accounting period. It will alert administrators and reinitiate the entire tuning process if it detects sufficiently large deviations. Note that such automatic detection of deviation may not be appropriate in some cases; e.g., in workloads with well-known diurnal patterns, but whose characteristics may change significantly throughout the day. In these cases, it is possible for OpTune to "remember" different profiling data and configurations appropriate for different periods during the day. One of our evaluation systems, a Hadoop MapReduce cluster, has some of these characteristics. The current implementation uses a mix of manual intervention and simple load prediction. We leave the design and implementation of more sophisticated mechanisms and approaches as future work.

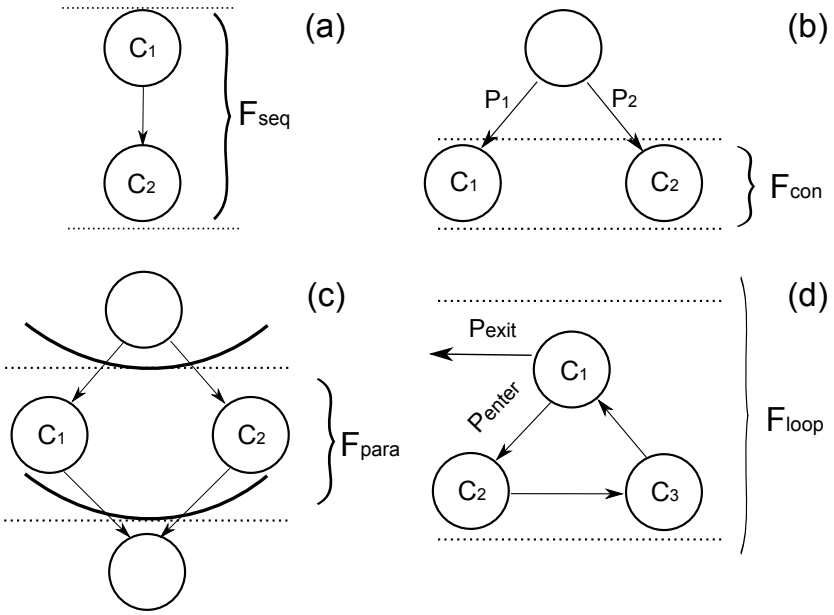


Figure 5.4: Sequential (a), conditional (b), parallel (c), and loop patterns.

5.2.2 Performance Composition

Given the service graph of a system, we view the system's service time as a random variable with a distribution that can be calculated if we know the transition probabilities of the edges and the distribution of the service time at each node. In general, most service graphs can be defined using a small number of patterns. We briefly discuss operations used to compose the CDFs for the basic patterns that are used to define the service graphs for the OpTune systems that we have built (Section 5.3).[1] We refer the reader to [119] for a more detailed discussion of a similar approach for predicting the performance of composed Web services.

Sequential. In a sequential pattern, execution always passes from a node to the one following it as shown in Figure 5.4(a). The distribution function that describes the performance of both components is then given by the convolution ($\circledast$) of the distributions of the components: $F_{seq}(t) = F_1(t) \circledast F_2(t)$, where F_i is the CDF for component Ci (equivalently $F_{seq} =$

[1] The composition depends on the fact that service times at different nodes are independent (i.e., not correlated). This assumption does not always hold. In Section 5.2.4, we explain how we implement the composition operationally so that we can account for correlations when needed.

$\int_0^t F_1(t-x)\,f_2(x)dx$, where $f_2(x)$ is the probability density function of $F_2(t)$).

Conditional. In a conditional pattern (Figure 5.4(b)), execution passes from a node to one of a set of following nodes according to the probability associated with each outgoing edge. The distribution function that describes the execution time of the set of following nodes is then given by a weighted sum ($\odot$) of the distributions of the components: $F_{con}(t) = F_1(t) \odot F_2(t) \odot ... \odot F_n(t)$, which can be computed as $F_{con}(t) = \int_0^t (\sum_{i=1}^n P_i f_i(x))dx$, where P_i is the probability for transitioning to node C_i. Note that the transition probabilities are required for this operation, but are left out of the notation for simplicity.

Parallel with synchronized merge. In this pattern (Figure 5.4(c)), execution passes from a node to the parallel execution of the set of following nodes, with a barrier at the end. The distribution function that describes the execution time of the set of parallel nodes is given by the product ($\otimes$) of the distributions of the components: $F_{par}(t) = F_1(t) \otimes F_2(t) \otimes ... \otimes F_n(t)$, which can be computed as $F_{par}(t) = \prod_{i=1}^n F_i(t)$.

Loop. In this pattern (Figure 5.4(d)), the execution loops through a number of states for a number of iterations before exiting the pattern. The distribution of the execution time of this pattern depends both on the distribution of the sub-graph within the loop, as well as the probability for the execution of another iteration vs. that of exiting the loop. The computation of this distribution is more involved, and so we refer the reader to [119] for the details. We denote this composition as: $F_{loop}(t) = \oplus(F_1, F_2, ..., F_n)$ where F_i is the performance CDF of component C_i in the loop. Similar to the $\odot$ operation, the transition probabilities are important to the computation but left out of the notation for simplicity.

Computing the composed CDF of a graph. Given a graph, we first compute the composed CDFs of loops; for nested loops, we start from the innermost loop and proceed outward. We then apply the remaining operations from the root (left) to the end nodes (right).

5.2.3 Performance Decomposition

OpTune must also address the reverse problem in order to find suitable configurations. That is, given a service graph and different transition probabilities and performance CDFs for each component corresponding to different settings of configuration parameters, how can OpTune choose the appropriate configuration parameter values to best meet a set of performance objectives. Our approach is to pose this as an optimization problem and then solve it.

Specifically, administrators can specify performance objectives as a set of points on a target performance CDF F_g, and optionally constraints such as $F_g^{-1}(99) < 200ms$. The optimization problem is then posed as the minimization of the mean squared error (MSE) between a solution CDF F_s and the target F_g:

$$\min \frac{1}{|P|} \sum_{p \in P} (F_g^{-1}(p) - F_s^{-1}(p))^2 \tag{5.1}$$

subject to the specified constraints, where P is the range (percentages) specified for F_g, and F^{-1} is the inverse of F (typically called the quantile function).

Alternatively, administrators can specify performance objectives as the minimization of a function $g()$ applied to the solution CDF F_s, and optionally constraints such as $F_g^{-1}(99) < 200ms$. In this case, the optimization problem is posed as:

$$\min \; g_{p \in P}(F_s^{-1}(p)) \tag{5.2}$$

where $g()$ can be an arbitrary function that produces a single value and P is a set of percentages specified by administrators. Average is a commonly used function; e.g., minimize the average response time, while ensuring that the 99^{th} percentile is less than 200ms.

As already mentioned, to solve for F_s, we divide each accounting period into E epochs of equal length (e.g., 2 minutes). We assume that the offered load is stable over the accounting period; note that this typically implies that the accounting period should be relatively short

(since the offered load is more likely to change over longer periods of time). We then search for a performance CDF F_e for each epoch e such that:

$$F_s(t) = \frac{1}{E}\sum_{e=1}^{E} F_e(t) \tag{5.3}$$

where each F_e is the system performance CDF given by a particular setting of configuration parameters C_e. Each of the CDFs F_e is computed by composing the component CDFs corresponding to C_e using the system service graph as explained in Section 5.2.2. The goal of course is to find the set of F_e that leads to an overall F_s giving the minimum for the posed optimization problem.

5.2.4 Implementing OpTune

We have built a prototype OpTune framework that can be integrated with different server systems. Some relevant aspects of the implementation are as follows.

Configuration parameters independence. Our approach of profiling components' performance at different settings of their configuration parameters, and then composing components' CDFs to predict overall service performance, is most efficient if the impact of parameters are localized and relatively independent. As an example, consider a system with n components $C_1, C_2, ..., C_n$, and m parameters $p_1, p_2, ..., p_m$. If all m parameters significantly impact all transition probabilities and/or the performance of all components, then OpTune would need to profile the system for $O(\prod_{i=1}^{m} v_i)$ different configurations, where v_i is the number of values that p_i can take on. On the other hand, if each parameter mostly impacts the transition probabilities and performance of a non-overlapping sub-graph of the service graph, then OpTune would need to profile the system for only $O(\sum_{i=1}^{m} v_i)$.

Currently, OpTune relies on administrators to identify the edges and components that are significantly impacted by each parameter. It uses these relationships to minimize the number of collected profiles. In future work, we will explore techniques for automatically detecting these

relationships.

Composing CDFs. Section 5.2.2 describes a mathematic for manipulating CDFs that is convenient for discussing how the transition probabilities and performance CDFs of components can be composed to predict the overall service performance. However, our implementation uses a sampling approach to implement the composition operations. This approach works as follows. When profiling the performance of a component, OpTune records a large number of execution times across a large number of a component's execution. This defines the component's performance CDF at a particular configuration setting. Then, to compute $F_1 \circledast F_2$, we would repeatedly compute a value of the resulting CDF by adding two values v_1 and v_2, chosen randomly from the set of execution times comprising F_1 and F_2, respectively. Other operations are implemented in a similar manner. Our profiling runs need to track the probabilities for different numbers of iterations across loop executions to support this implementation.

The above approach allows us to account for correlation between the performance of different components. For example, supposed the execution times of two components C_1 and C_2 are correlated. Then, when computing $F_1 \circledast F_2$ (F_1 is C_1's performance CDF), we will choose v_1 and v_2 in a manner that respects the correlation. This was not needed in any of the three systems we implemented. When composing to predict the performance of a service, we ensure that there are enough sampling points so that compositions are always statistically significant [74, 61].

Solving the OpTune optimization problem. Currently, our implementation performs a complete search over the possible configuration settings to find the best solution to the optimization problem. This approach works well for a small number of configuration parameters; for example, solving the optimization problem for the three server systems that we implemented and evaluated, each with two parameters exposed to OpTune, always took less than 40 seconds. In the future, we intend to explore a more scalable approach based on a search heuristic (e.g., simulated annealing).

User interface. It is possible that no solution exists that satisfy the constraints specified by

administrators, or that the best fitting solution is very different from the target CDF. In this case, administrators can iterate through multiple runs of OpTune, modifying the performance objectives. To ease this task, OpTune can show different CDFs that can be achieved using different configuration settings.

5.3 OpTune Systems

We have built three OpTune systems using the above framework, including a Web server, a filesystem emulator, and a MapReduce scheduler. We describe the design and implementation of these systems in this section.

5.3.1 Web Server

We have modified the Mongoose Web server [4] to work with OpTune. The primary performance metric for this server is request processing time. Figure 5.2 shows the service graph for the Web server comprising four components. We modified Mongoose to measure the execution time of the code corresponding to these components. The Web server's CDF of request processing time (F_{WS}) can then be computed as: $F_{WS} = F_1 \oplus (F_2 \odot F_3) \oplus F_4$, where F_i is the performance CDF of component C_i.

Two configuration parameters are exposed to OpTune for performance tuning. As shall be seen, we choose these two parameters because their settings can strongly shape the entire performance CDF of the server. The first is the `idle_time` parameter in the disk I/O layer (inside the Linux kernel). This parameter specifies the length of time that the CFQ I/O scheduler will wait for another request from a thread it is currently servicing before switching to servicing I/O requests from another thread. It has been observed that in many cases, a stream of I/O requests from a single thread will correspond to sequential accesses to files. This is especially true for a Web server. Thus, increasing `idle_time` can reduce disk head movement, improving both throughput and average response time [10, 51]. However, it can increase the tail response

time if a request is delayed while the I/O scheduler switches through several other threads. As shall be shown below, this parameter's setting can have a strong impact on the CDF of response time for disk requests, which in turn has a significant impact on the Web server's performance CDF.

The second configuration parameter is in the caching subsystem. Caching inside Web servers and proxies have been studied extensively [19, 26, 22, 5], and it has been shown that accounting for factors such as temporal locality, popularity, and size is important for maximizing performance. However, accounting for these factors represent tradeoffs. For example, in some cases, it may be desirable to achieve the highest hit rate, since this corresponds to the highest percentage of clients experiencing low response time. On the other hand, high hit rates disproportionately favors the caching of small files. Thus, in other cases, it may be more appropriate to maximize the byte hit ratio, which trades off more misses for small files to get the benefits from caching larger files.

In our Web server, we use the GDSF algorithm, which has been shown to work well [19, 26].[2] This replacement algorithm considers three different metrics for choosing victims for eviction when there is a miss and the cache is full. These metrics include aging based on time of last access, frequency of access, and object size. Briefly, GDSF works as follows. Each cached file is assigned a priority $P(f)$ computed as:

$$P(f) = clock + Freq(f)\frac{Cost(f)}{Size(f)} \tag{5.4}$$

When there is a miss, and the cache does not have enough free space to cache the referenced file, the set of files with the lowest priorities are evicted to make space. When a file is first brought into the cache, $Freq(f) = 1$ and $Size(f)$ is a function of f's size in bytes. Whenever there is a hit for f in the cache, $Freq(f) = Freq(f) + 1$, and f's priority is recomputed.

[2] We chose to use Mongoose because it is a mature server that has been available since 2004, yet is relatively easy to modify. However, Mongoose did not include a memory cache, which is critical for performance in many production environments. Thus, we added a memory cache to Mongoose.

Whenever a set of files $f_1, f_2, ..., f_n$ are evicted, $clock = \max_{i=1}^{n} P(f_i)$.

There are actually four possible configuration parameters for tuning GDSF's performance, the function for increasing clock, the $Size(f)$ function, the $Cost(f)$ function, and the function for increasing $Freq(f)$. For simplicity, we focus on $Size(f)$ as the performance tuning knob because we find that it gives the largest trade-off between performance and variability; others have found size to be a critically important parameter as well, e.g., [6].

5.3.2 Filesystem Emulator

We have also integrated OpTune into a filesystem emulator. This emulator takes a trace of I/O requests and a file-to-disk-block mapping, and emulates the servicing of the I/O requests. The emulator emulates the operation of a filesystem by implementing a block-based buffer cache on top of a disk partition. It accesses the disk for cache misses using direct I/O to bypass the OS buffer cache. It services each I/O request by computing the set of blocks needed using the file-to-disk-block mapping, and then retrieves the blocks from the buffer cache or disk as appropriate. Writes are buffered in the buffer cache, and written back to disk by a background flusher thread. Note that the Linux kernel writes back dirty blocks based on several conditions (current load, flush threshold, etc.). For simplicity, our implementation flushes dirty blocks periodically (every 30 seconds) or when the number of dirty block reaches a high watermark.

Figure 5.6 shows a high-level service graph for this system. Writes of dirty blocks are not on the critical path (sufficient free space is maintained so that a write is never necessary on the eviction of a dirty block), and so are not included in the service graph. C_2 and C_3, which handle hits and misses in the buffer cache, respectively, form a multi-loop structure. An I/O request containing n blocks, with m hits and $n-m$ misses, would loop through C_2 m times and C_3 $n-m$ times. Disk accesses are performed using asynchronous I/O. C_4 includes the wait time for all disk accesses to complete. The filesystem emulator's CDF of request processing time (F_{FS}) can then be computed as:

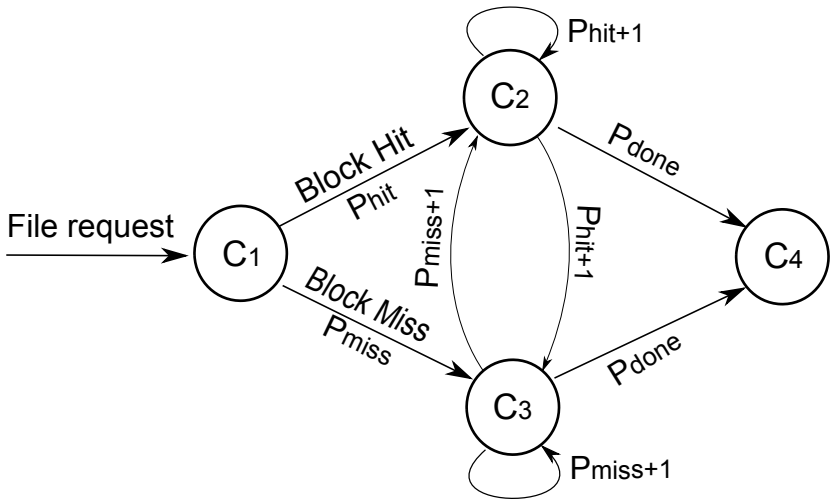


Figure 5.5: File server composition graph. C_1: process file request, C_2: get requested block from buffer cache, C_3, service missed block from disk, C_4: wait for all disk accesses to complete, complete processing of request, and return.

$$F_{FS} = F_1 \circledast ((\oplus(F_2, F_3) \circledast F_4) \odot (\oplus(F_3, F_2) \circledast F_4))$$

Two configuration parameters are again exposed to OpTune for performance tuning because of their strong impact on the system's performance CDF. The first is the same `idle_time` parameter in the disk I/O layer already described above. The second is used to move between an LRU replacement strategy and one based on popularity/frequency. This parameter is motivated by previous work showing that no single replacement policy is best for all possible workloads/environments (e.g., [9, 80]). Specifically, we implement a configuration parameter called `eviction_prob`, which takes on a value between 0 and 1. When it is set to 1, then a victim chosen for replacement using an LRU replacement policy is always evicted. When set to 0, cached items are never evicted. Thus, if the cache is filled with the most popular items, this policy emulates a popularity-based caching scheme. When `eviction_prob` is set between 0 and 1, then the victim will be evicted with probability equal to `eviction_prob`. This increases the chances for popular items to be removed from the cache as they are not used for longer periods of time.

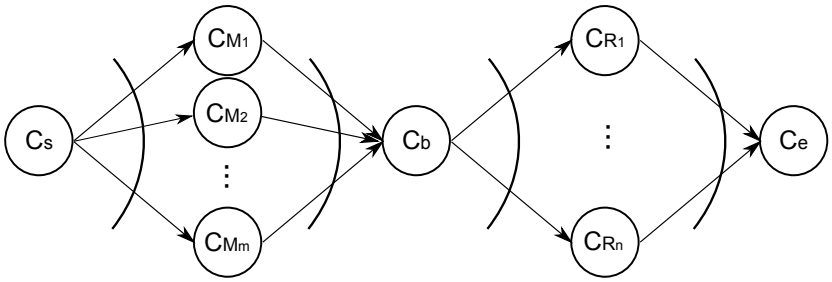


Figure 5.6: MapReduce composition graph. C_s: preprocess and schedule job, $C_{M1},...,C_{Mm}$: perform map tasks, C_b: barrier between Map and Reduce phases, $C_{R1},...,C_{Rn}$: perform reduce tasks, C_e: complete job.

5.3.3 MapReduce System

Finally, we have integrated OpTune with the Hadoop MapReduce scheduler to explore its behavior in a system that is drastically different from the Web and filesystem servers. In this system, we use OpTune to tune the CDF of job completion times. Figure 5.6 shows the service graph for completing a Hadoop MapReduce job. Each job has two phases, a Map phase where all map tasks ($C_{M,1}, C_{M,2}, ..., C_{M,m}$) are executed and a Reduce phase where all reduce tasks ($C_{R,1}, C_{R,2}, ..., C_{R,n}$) are executed. The job is initiated in C_s. C_b transitions between the Map and Reduce phases, and C_e saves the output and completes the job.

Given this service graph, the performance CDF of the system (F_{MR}) can be computed as:

$$F_{MR} = F_s \oplus (F_{M1} \otimes ... \otimes F_{Mm}) \oplus F_b \oplus (F_{R1} \otimes ... \otimes F_{Rm}) \oplus F_e$$

We tune the performance of this system by adjusting the scheduling policy and dropping the execution of a subset of map tasks. For the scheduling policy, we implemented a parameter `prob_SJF` that moves the scheduling policy between FIFO, which gives better fairness since jobs are executed in order of arrival, and Shortest-Job-First (SJF), which reduces average waiting time but may starve large jobs. `prob_SJF` allows a mix of the two scheduling policies, allowing the administrator to favor one over the other by sliding the parameter. Note that Hadoop has been specifically implemented to allow configuration with different pluggable schedulers. In this case, we are introducing/studying a scheduler that allows dynamic tuning,

rather than the static *a priori* selection of a single scheduler.

We implement the mixed scheduling policy using Hadoop's job priorities 1-5, with 1 being lowest and 5 being highest. When a job arrives, we randomly determine whether FIFO or SJF should be used based on `prob_SJF`. If FIFO, then the job is given priority 3. If SJF, then the job is given a priority based on the number of reduce tasks (which we use as a rough estimate of job size). The partitioning between priorities is such that jobs with sizes around the median are given priority 3, the largest priority 1, and the smallest priority 5.

The second configuration parameter `drop_p_maps` allows the administrator to trade precision for reduced completion times. In particular, this parameter controls whether map tasks can be dropped from the execution of a MapReduce job. When non-zero, `drop_p_maps` percent of map tasks are randomly chosen and dropped from the execution of each job. While we are introducing this parameter, we note that dropping tasks has been used to enable approximations in MapReduce with small inaccuracy bounds, as shown in [43, 97]. Thus, we hypothesize that parameters for controlling approximation similar to `drop_p_maps` will be introduced into future approximation-enabled MapReduce frameworks.

5.4 Evaluation

We now turn to explore and evaluate OpTune's efficacy at helping administrators to tune their systems to achieve specific performance goals. We present mostly results from the Web server, although we also show some results for the MapReduce system and filesystem emulator toward the end of the section.

5.4.1 Experimental Setup

Experimental platform. Experiments for the Web and filesystem servers were run on a server machine equipped with a 2.4 GHz 4-core, each with 2 hyper-threads, Xeon CPU, 8 GB of RAM, and a 160 GB 7200 RPM SATA hard disk. The server was running Linux 3.2.54, with

the scheduling policy of the disk I/O subsystem set to Completely Fair Queuing (CFQ).

Experiments for the MapReduce system were run on a 10-machine cluster, where each machine is equipped with a 1.8 GHz 2-core, each with 2 hyper-threads, Opteron CPU, 8 GB of RAM, and a 750 GB 7200 RPM SATA hard disk. The servers were running Hadoop 1.1.2 on top of Linux 2.6.18. We modified the Hadoop `JobQueueTaskScheduler` class to assign priority to jobs based on the size of their input data. The system was configured with 40 map slots and 10 reduce slots (4 map slots and 1 reduce slot for each server).

Web server workload. We use ProWGen [17] to generate a Web access trace. We use the default Zipf distribution with parameter 0.9 and Pareto distribution with tail index of 1.2 to model object popularity and object size, respectively [17]. The median object size is set to 60 KB with standard deviation of 10 MB based on studies of previous Web server workloads [42, 52, 84]. Finally, requests arrive according to a Poisson process with mean inter-arrival time of 72ms, leading to an average utilization of approximately 50%. We generate a trace lasting 4 hours, using the first 2 hours for profiling and system warmup and the last 2 hours for our experiments.

Filesystem workload. We use a trace from the Microsoft Production Server Traces [55, 101]. Traces in this set were collected from a number of different Microsoft production servers, and include information such as process ID, operation type, file descriptor, offset, and size. These traces contain sufficient information for us to build the needed file-to-disk-block mapping. We use the 6-hour MSN Storage File Server trace to study the behavior of typical file servers. We use the first 4 hours for profiling and system warmup and the last 2 hours for our experiments. The trace was collected from a more powerful server than our, so we slowed the trace down such that average throughput is approximately 60% of saturation.

MapReduce workload. We use the Statistical Workload Injector for MapReduce (SWIM) [25] to generate a scaled-down 6-hour workload from a larger Facebook trace collected from May to October 2009. In the resulting workload, each job comprises 2-120 map tasks and 1-20 reduce

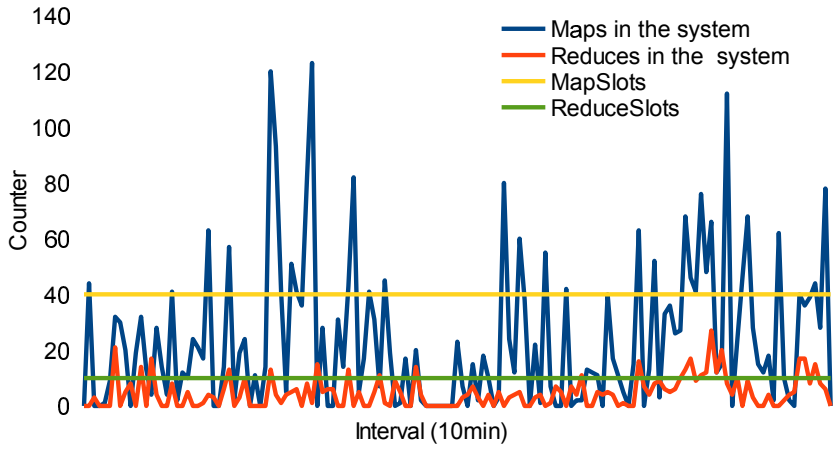


Figure 5.7: The SWIM trace are classified into different phases based on utilization. Task execution and queueing times are sampled from different utilization phases accordingly.

tasks. There are ~700 jobs with ~8000 tasks. The map phase of each job takes 50-300 seconds, and the reduce phase takes 15-100 seconds. Jobs have inputs of 64MB-9GB and outputs of up to 1GB. Figure 5.7 plots this workload, which gives an average cluster utilization of 64%. We use the first 3 hours for profiling and the last 3 hours for our experiments.

Recall that the completion times of map and reduce tasks include wait times, which are different at different load. Thus, we define five different load levels as follows: VERY_LOW (<5 active map tasks), LOW (<10), NORMAL (<15), HIGH (<50) and VERY_HIGH (≥50). OpTune then collects a set of profiling information for each different load level, and solves the optimization problem separately for each load level to get configuration settings that best meet the performance objectives for that level. At runtime, OpTune predicts the load level at the beginning of each accounting period to be the same as that observed in the accounting period that just completed. It then configures the system according to the predicted load level.

5.4.2 Impact of Configuration Parameters

Impact of caching size priority parameter. We begin our study by exploring the impact that different values of configuration parameters can have on the performance of a system. Figure 5.8 plots the performance CDFs for the Web server when we set the I/O idle-time parameter

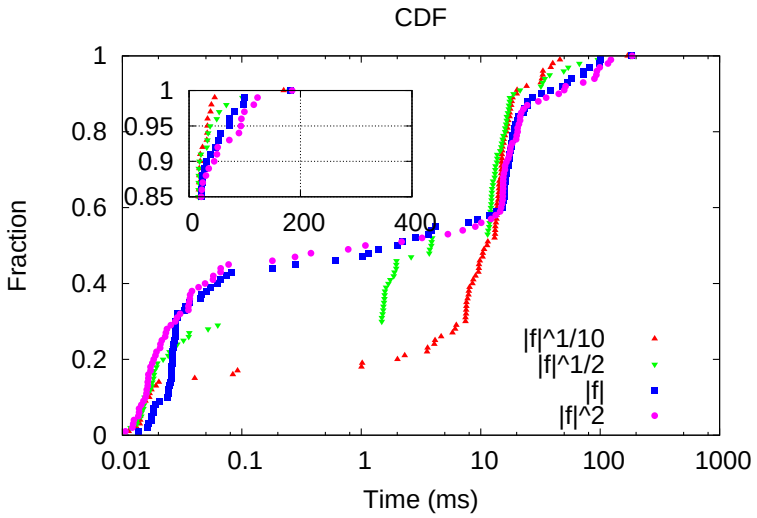


Figure 5.8: Impact of $Size(f)$ on the Web server's response time.

to 8ms and the function $Size(f)$ to $|f|^{\frac{1}{10}}, |f|^{\frac{1}{2}}, |f|, |f|^2$, where $|f|$ = *number of bytes in* f.[3] These functions lead to 18%, 24%, 41% and 44% item-wise hit-ratios, respectively. It is easy to observe that lowering the caching priority of larger objects (e.g., $Size(f) = |f|^2$) can substantially improve response time for a fraction of the requests (differences in CDFs from ~15-45%). This is because larger objects will push out smaller objects less frequently, leading to more effective caching for the smaller objects. On the other hand, favoring the smaller objects can significantly increase the tail response time, as requests for the largest objects will most likely lead to cache misses.

Impact of I/O `idle_time` parameter. We next explore the impact of the I/O idle-time parameter on server performance. Specifically, we set the caching $Size(f)$ parameter to $|f|$, which gives a hit ratio of 41%, and we set the idle-time (I) in CFQ to 0ms, 4ms, 8ms, 16ms, and 24ms. Figure 5.9 plots the results. Again, it is easy to observe that the parameter value offers tradeoffs between the response time for a significant portion of the CDF (between ~50%-90%) against the tail (>95%).

[3] We also adopt the common approach of not caching files larger than 2MB to avoid polluting the cache with very large objects [22, 3, 5].

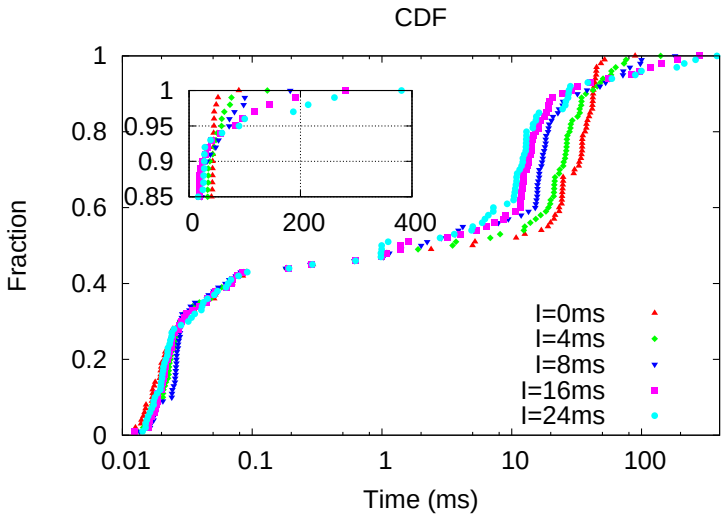
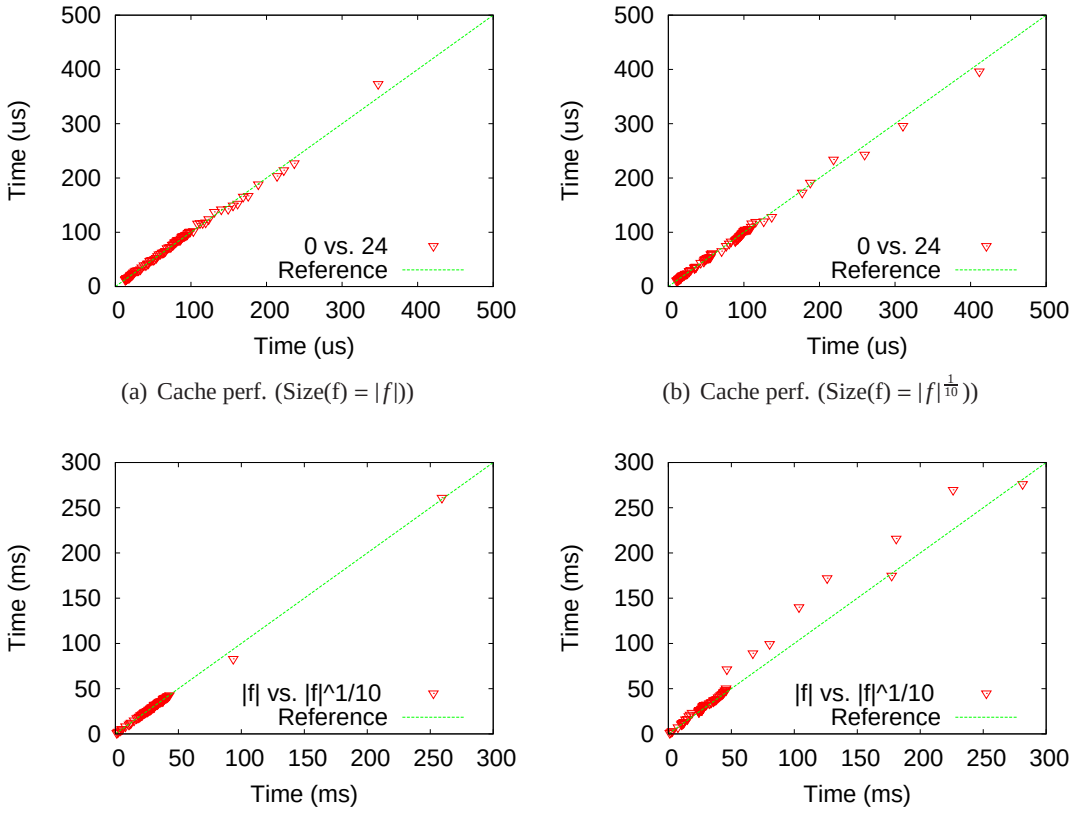


Figure 5.9: Impact of `idle_time` on the Web server's response time.

Configuration parameters independence. We specify to OpTune that $Size(f)$ affect P_{hit}, P_{miss}, and the performance of C_2, while `idle_time` affect the performance of C_3. In Figure 5.10, we study the accuracy of this information. Specifically, Figure 5.10(a) and (b) show that C_2's performance CDF remains almost the same even when `idle_time` is set to two very different values. Thus, `idle_time` indeed does not impact C_2's performance. On the other hand, Figure 5.10(c) and (d) show that the tail of C_3's performance CDF is affected somewhat by $Size(f)$ when `idle_time` = 24ms. We deemed the inaccuracies introduced to be small enough that it was acceptable to assume C_3 is independent of $Size(f)$ to keep profiling overheads low.

Figure 5.11 shows an example of the potential inaccuracies arising from OpTune's various assumptions. The figure shows a target CDF chosen to highlight inaccuracies introduced by the specified independence assumptions (the long tail), the CDF predicted by OpTune for its chosen configuration(s), and the resulting observed CDF when the server was run with OpTune's chosen configuration(s). Observe that while there are some inaccuracies, the fit between predicted and actual is quite good.



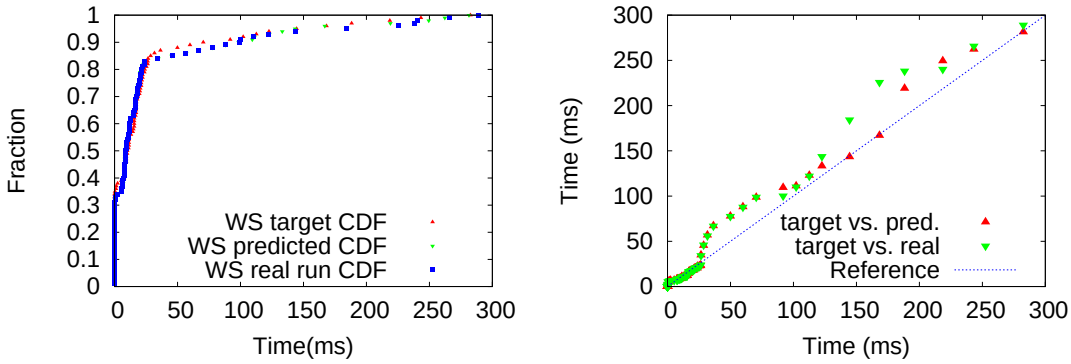
(a) Cache perf. (Size(f) = $|f|$)) (b) Cache perf. (Size(f) = $|f|^{\frac{1}{10}}$))

(c) File read perf. (I=0ms) (d) File read perf. (I=24ms)

Figure 5.10: Quantile-quantile plots for (a-b) comparing cache access time (performance of C_2) when `idle_time` is set to 0ms vs. 24ms for (a) $Size(f) = |f|$ and (b) $Size(f) = |f|^{\frac{1}{10}}$; (c-d) comparing file read times (performance of C_3) when $Size(f)$ is set to $|f|$ vs. $|f|^{\frac{1}{10}}$ for (c) `idle_time` (I) = 0ms and (d) `idle_time` = 24ms;

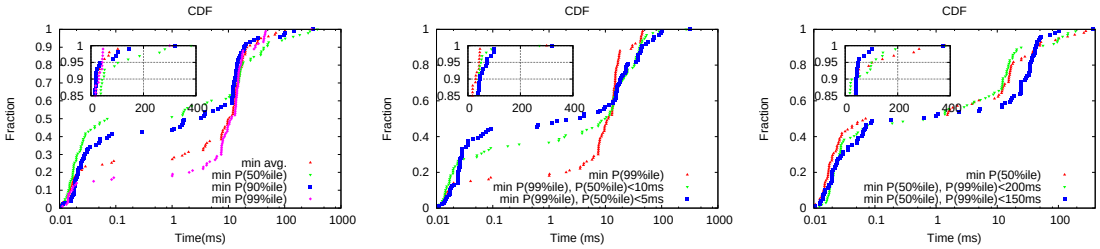
5.4.3 Performance Tuning

Single point performance optimization. We begin by studying the impact of optimizing for a single point on the CDF. Figure 5.12(a) shows this single point optimization when OpTune optimizes for the smallest average, median, 90^{th} percentile, and 99^{th} percentile. Table 5.1 lists the average, median, 90^{th} percentile, and 99^{th} percentile response times for each of these optimization goal. Observe that depending on whether the user is more concerned with average/median performance or the tail, the overall performance CDF differs significantly. Specifically, when we are interested in minimizing the 99^{th} percentile, OpTune sets $Size(f)$ to $|f|^{\frac{1}{10}}$ and $idle-time$ to 0. These settings lead to the smallest 99^{th} percentile response time (46ms), but degrades



(a) CDFs (b) QQ-plots

Figure 5.11: Accuracy of OpTune's performance prediction for an example target Web server performance CDF.



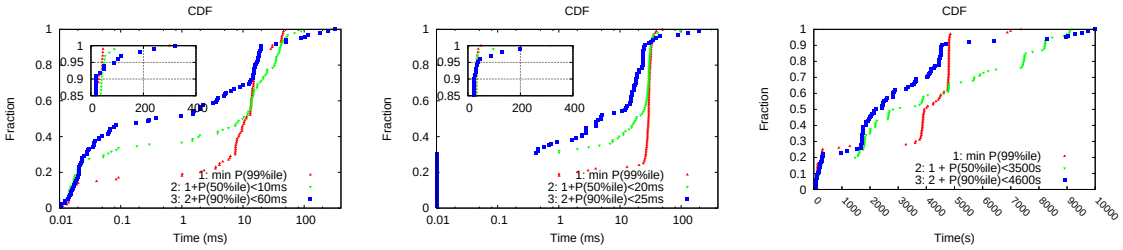
(a) Single point optimization (b) Minimizing tail with constraint for median (c) Minimizing median with constraint for tail

Figure 5.12: Web server's response time for different performance objectives.

performance significantly for lower percentiles (e.g., median response time of 11.3ms). In contrast, optimizing the median leads OpTune to set $Size(f)$ to $|f|^2$ and $idle-time$ to 24ms. These parameters lead to a much lower median response time (0.1ms), but significantly degrades the 99^{th} percentile latency (281ms). Minimizing the average leads to longer response time for shorter requests (median time of 10.4ms), but significantly smaller 99^{th} percentile time (102.2ms).

Goal	**Avg.**	**Median**	**90^{th}-%ile**	**99^{th}-%ile**
Min Avg.	12.1	10.4	24.2	102.2
Min Median	22.4	0.1	49.6	281.3
Min 90^{th}-%ile	13.4	3.2	19.5	143.7
Min 99^{th}-%ile	13.9	11.3	30.1	46.0

Table 5.1: Detailed results for web server single point optimization. Times are given in ms.



(a) Web server (b) File server (c) MapReduce

Figure 5.13: Server response time for different numbers of constraints.

Multi-point performance optimization. As discussed previously, it is frequently not desirable to optimize for a single point of performance. Rather, the user may want to realize multiple performance goals, such as minimizing the 99^{th} percentile response time, while maintaining a target median response time. Meeting such performance goals is exactly what we set out to do with OpTune.

Figure 5.12(b) shows the results when the user wants to minimize the 99^{th} percentile response time while constraining median response time to be no worse than 10ms and 5ms. Observe that as the bound for the median response time becomes tighter (10ms $\rightarrow$ 5ms), OpTune has to trade off a progressively "longer" tail for the desired median performance.

Figure 5.12(c) shows that OpTune is also effective when the constraint and optimization goal are exchanged; that is, in these cases, OpTune is seeking to minimize the median performance while observing a constraint for the 99^{th} percentile.

Figure 5.13(a) shows the results when minimizing the 99^{th} percentile performance with bounds on the median and the 90^{th} percentile performance. Interestingly, when we introduce the bound for the 90^{th} percentile performance, the tail becomes much worse, while the median becomes much better compared to when we only bound the median. This demonstrates OpTune's ability for full-range performance tuning, and the difficulty facing administrators without OpTune when performance objectives require thinking about multiple points on the performance CDF.

Figures 5.13(b) and 5.13(c) show the results when minimizing the 99^{th} percentile performance with bounds on the median and the 90^{th} percentile performance for the filesystem emulator and MapReduce system. Results are similar in that adding performance objectives (constraints in these cases) can lead to significant changes over the entire performance CDF. Such full-range tuning would be very difficult for administrators to manage manually.

Full performance CDF target. We previously showed results in Figure 5.11 for OpTune seeking to meet performance objectives specified as a full CDF curve (100 points).

5.4.4 Sensitivity Analysis

We have studied the sensitivity of our results for the Web server to different workload characteristics, including load intensity, correlation between object size and popularity, and the distribution of object sizes. As these characteristics change, the tradeoffs embodied in the configuration parameters can increase or decrease. For example, Figure 5.14 shows that a higher load intensity can significantly increase the tradeoffs given by different settings of `idle_time`. Correspondingly, results for a lower load intensity (not shown here) shows less tradeoffs. Overall, we find that the parameters exposed to OpTune for the three systems continue to embody significant tradeoffs across a wide range of different workload characteristics. Thus, we conclude that OpTune should be widely applicable to full-range performance tuning of many server systems and many different workloads.

5.5 Summary

In this chapter, we proposed and evaluated OpTune, a framework for helping administrators to configure server systems to best achieve a set of performance objectives. Administrators can use OpTune to find settings for multiple interacting configuration paramaters in order to shape the entire performance CDF of a server system. Administrators can also use OpTune to ask what-if questions. For example, what will happen to the performance CDF of a system if its

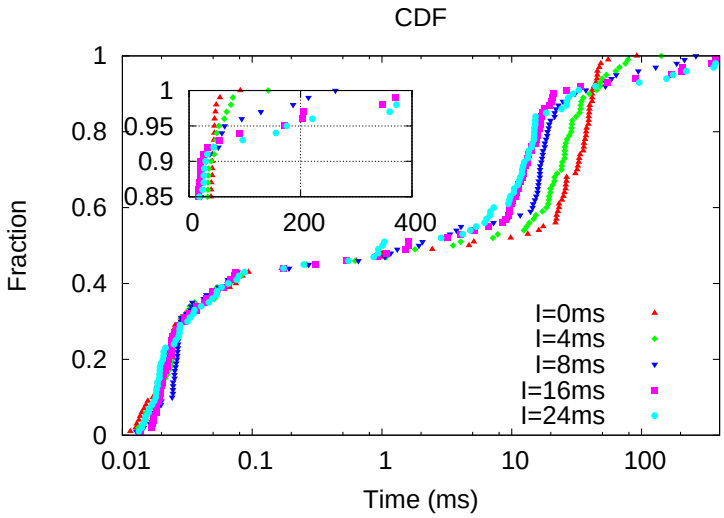


Figure 5.14: Impact of I/O `idle_time` on the Web server's performance for high load.

configuration is modified to meet an additional constraint such as "make the 90^{th} percentile response time less than X." Such tasks are extremely difficult to perform manually, and will only become more difficult as server systems becomes ever more complex.

We have integrated OpTune into three different server systems: a Web server, a file system emulator, and a MapReduce scheduler. Our evaluation results show that configuration parameters embody significant tradeoffs between different parts of the performance CDF—e.g, configuring to reduce tail response times can significantly worsen response times for shorter requests—and that OpTune is a powerful tool for helping administrators explore such tradeoffs and configure systems appropriately for performance objectives driven by different needs.

Chapter 6

Conclusion and Future Work

In this dissertation, we addressed three key performance challenges in server systems: the inefficiency in SSD cache space utilization, the performance interference and variability in virtualized systems, and the desire to meet multiple performance targets at the same time. We explored these challenges concretely by designing, implementing, and evaluating three systems: Nitro, VirtualFence, and OpTune.

In Nitro, we leveraged data reduction techniques to improve cost efficiency of a HDD/SSD hybrid storage system while achieving high performance. In VirtualFence, we leveraged SSDs and non-work-conserving scheduling to provide consistent I/O performance for virtualized environment. In OpTune, we leveraged manipulation of performance CDF to manage multi-target performance for server systems. We used optimization to solve configurations for different performance targets. Our evaluations showed that our systems can achieve specified performance targets for these performance tuning tasks. In most cases, we achieve desirable performance targets based on user's requirements.

Finally, we conclude that the techniques proposed in this dissertation shows great potential in performance management in server systems. The notion of optimizing performance while considering other constraints can be used in other system research works, which has had a significant impact on both industry and academia. As server systems become more complex and cloud computing becomes more popular, performance improving and tuning will confront an even greater challenge. In fact, the definition of best performance will not be easily determined because desirable performance is different across various scenarios. Our systems are strong

steps toward effectively managing the performance of modern server systems. For example, providing predictable performance for tenant VMs will become challenging as the scale of cloud service increases and/or the workload types increase. The heterogeneity and overhead of tuning performance may also become a concern for cloud providers who want high consolidation and high performance. Further research along these lines will lead to promising outcome. As for future work, we propose several possible directions:

Caching algorithm for WEU. Although Nitro [70] optimizes overall storage system performance while minimizing cost by leveraging coarse-grained WEU, there is limited work to systematically study the benefit of WEU and co-design a cache eviction algorithm that maximizes WEU's benefit on performance and lifespan. Classic caching algorithms leverage recency, frequency, and/or other properties of cached blocks at per-block granularity. However, WEU comprises multiple logically distinct, but physically co-located, blocks. WEU may have highly diverse blocks, with mixtures of frequently accessed, infrequently accessed, and invalidated blocks. A simple caching algorithm such as the WEU-based LRU might be insufficient to handle blocks with diverse access patterns [71]. Creating new policies for caching compound objects in flash remains an open research problem.

Extending predictable performance research in other systems. VirtualFence [68] shows promising results for virtualized environments in the organization of virtual machines. As container technology [98] becomes increasingly popular, containers might be a new medium to encapsulate resources as compared to virtual machines. A similar performance and resource management problem arises in the context of containers. Techniques used in VirtualFence such as time/space resource partitioning will continue to have an impact in the new context.

Automatic multi-point perform tuning. OpTune shows that tuning performance with multi-point constraints is a challenging task. Previous works such as ACI [121] and [89] have leveraged adaptive control techniques to perform automatic configuration management. We believe that further leveraging these techniques can be helpful for large Internet service providers, such

as Google and Amazon.

Appendix A

Additional Evaluation for OpTune

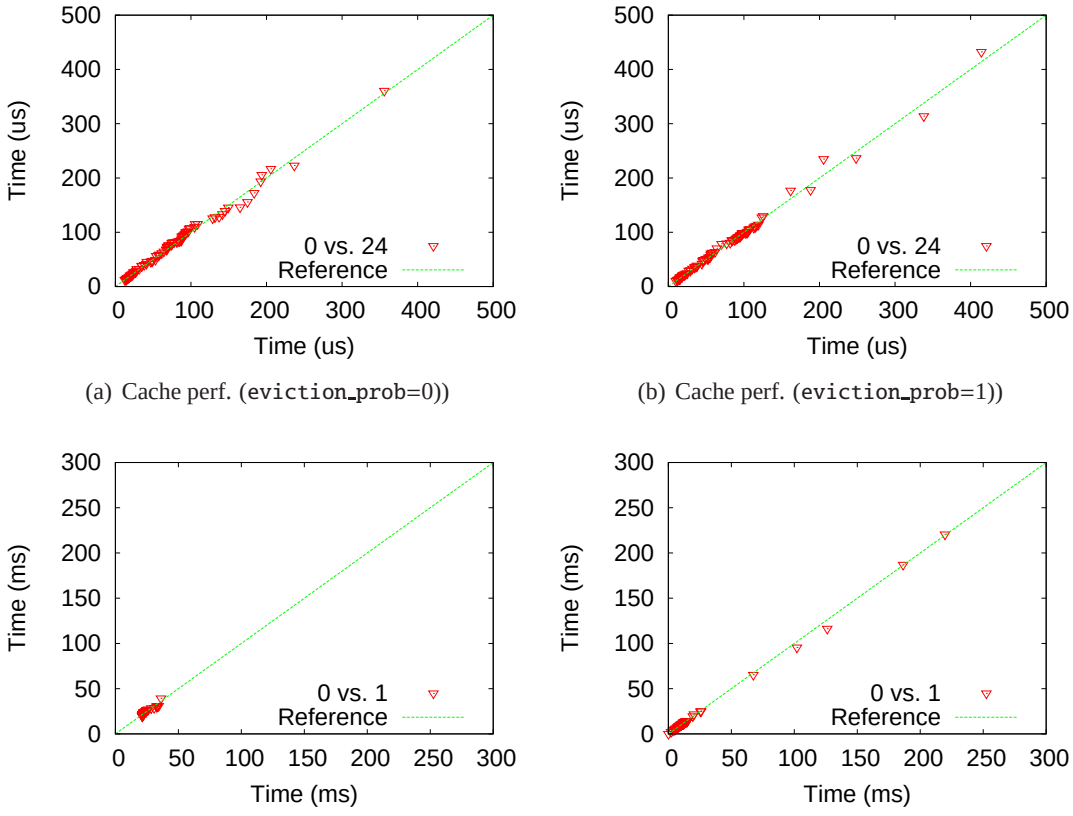
We first present the evaluation of the file system emulator and MapReduce system. Specifically, we study the impact of independence test, single point performance target tuning and multi-point performance targets tuning for each system. Next, we present overhead analysis for OpTune.

A.1 Filesystem Emulator with OpTune

First, we study applying OpTune to filesystem emulator. We study the independence test, single-point performance target tuning, multi-point performance targets tuning.

A.1.1 Independent Tests

We specify to OpTune that `eviction_prob` affect cache hit-ratios, while `idle_time` affect the performance of disk I/O. In Figure A.1, we study the accuracy of this information. Specifically, Figure A.1(a) and (b) show that C_2's performance CDF remains almost the same even when `idle_time` is set to two very different values. Thus, `idle_time` indeed does not impact C_2's performance. On the other hand, Figure A.1(c) and (d) show that the tail of C_3's performance CDF is affected somewhat by `eviction_prob` when `idle_time` = 24ms. The RMSE for both I=0ms and I=24ms are 0.4 and 1.3, which means the CDFs in one tier will not affect the CDF in another tier. We deemed the inaccuracies introduced to be small enough that it was acceptable to assume C_3 is independent of `eviction_prob` to keep profiling overheads low.



(a) Cache perf. (`eviction_prob`=0))

(b) Cache perf. (`eviction_prob`=1))

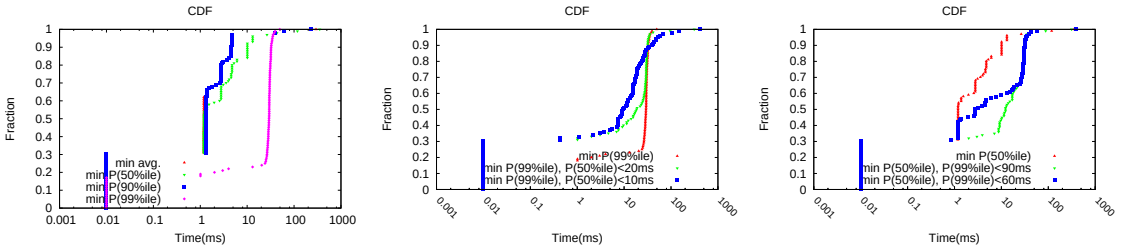
(c) File read perf. (I=0ms)

(d) File read perf. (I=24ms)

Figure A.1: Quantile-quantile plots for (a-b) comparing cache access time (performance of C_2) when `idle_time` is set to 0ms vs. 24ms for (a) `eviction_prob`=0 and (b) `eviction_prob`=1; (c-d) comparing file read times (performance of C_3) when `eviction_prob` is set to 0 vs. 1 for (c) `idle_time` (I) = 0ms and (d) `idle_time` = 24ms;

A.1.2 Single-point Performance Optimization

We begin by studying the impact of optimizing for a single point on the CDF. Figure A.2(a) shows this single point optimization when OpTune optimizes for the smallest average, median, 90^{th} percentile, and 99^{th} percentile. Table A.1 lists the average, median, 90^{th} percentile, and 99^{th} percentile response times for each of these optimization goal. Observe that depending on whether the user is more concerned with average/median performance or the tail, the overall performance CDF differs significantly. Specifically, when we are interested in minimizing the 99^{th} percentile, OpTune sets `eviction_prob` to 0 and `idle_time` to 0. These settings lead to the smallest 99^{th} percentile response time (39.4ms), but degrades performance significantly for



(a) Single point optimization

(b) Minimizing tail with constraint for median

(c) Minimizing median with constraint for tail

Figure A.2: Filesystem emulator's response time for different performance objectives.

lower percentiles (e.g., median response time of 29.3ms). In contrast, optimizing the median leads OpTune to set `eviction_prob` to 0.8 and `idle_time` to 24ms. These parameters lead to a much lower median response time (1.1ms), but significantly degrades the 99^{th} percentile latency (117.8ms).

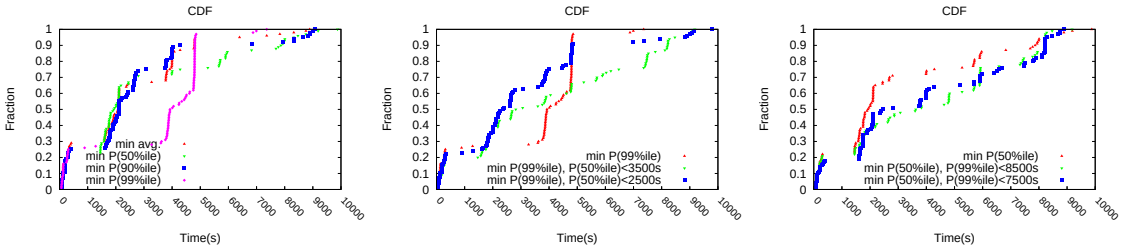
Goal	Avg.	Median	90^{th}-%ile	99^{th}-%ile
Min Avg.	5.6	1.2	7.56	61.2
Min Median	8.06	1.1	10.1	117.8
Min 90^{th}-%ile	5.9	2.8	6.5	61.2
Min 99^{th}-%ile	23.2	29.3	32.4	39.4

Table A.1: Detailed results for filesystem emulator single point optimization. Times are given in ms.

A.1.3 Multi-point performance optimization

As discussed previously, it is frequently not desirable to optimize for a single point of performance. Rather, the user may want to realize multiple performance goals, such as minimizing the 99^{th} percentile response time, while maintaining a target median response time. Meeting such performance goals is exactly what we set out to do with OpTune.

Figure A.2(b) shows the results when the user wants to minimize the 99^{th} percentile response time while constraining median response time to be no worse than 20ms and 10ms. Observe that as the bound for the median response time becomes tighter (20ms $\rightarrow$ 10ms), OpTune has to trade off a progressively "longer" tail for the desired median performance.



(a) MapReduce. (b) Minimizing tail with constraint for median (c) Minimizing median with constraint for tail

Figure A.3: MapReduce's response time for different performance objectives.

Figure A.2(c) shows that OpTune is also effective when the constraint and optimization goal are exchanged; that is, in these cases, OpTune is seeking to minimize the median performance while observing a constraint for the 99^{th} percentile.

A.2 MapReduce with OpTune

First, we study applying OpTune to MapReduce system. We study the independence test, single-point performance target tuning, multi-point performance targets tuning.

A.2.1 Single-point Performance Optimization

We begin by studying the impact of optimizing for a single point on the CDF. Figure A.3(a) shows this single point optimization when OpTune optimizes for the smallest average, median, 90^{th} percentile, and 99^{th} percentile. Table A.2 lists the average, median, 90^{th} percentile, and 99^{th} percentile response times for each of these optimization goal. Observe that depending on whether the user is more concerned with average/median performance or the tail, the overall performance CDF differs significantly. Specifically, when we are interested in minimizing the 99^{th} percentile, OpTune sets `prob_SJF` to 0 and `drop_p_maps` to 0.8. These settings lead to the smallest 99^{th} percentile response time (6993s), but degrades performance significantly for lower percentiles (e.g., median response time of 3924s). In contrast, optimizing the median leads OpTune to set `prob_SJF` to 1.0 and `drop_p_maps` to 0.9. These parameters lead to a

much lower median response time (2923s), but significantly degrades the 99^{th} percentile latency (9188s).

Goal	Avg.	Median	90^{th}-%ile	99^{th}-%ile
Min Avg.	2412	2079	4792	8756
Min Median	2923	1926	7864	9188
Min 90^{th}-%ile	2578	2103	4284	8996
Min 99^{th}-%ile	3360	3924	4807	6993

Table A.2: Detailed results for web server single point optimization. Times are given in second.

A.2.2 Multi-point performance optimization

As discussed previously, it is frequently not desirable to optimize for a single point of performance. Rather, the user may want to realize multiple performance goals, such as minimizing the 99^{th} percentile response time, while maintaining a target median response time. Meeting such performance goals is exactly what we set out to do with OpTune.

Figure A.3(b) shows the results when the user wants to minimize the 99^{th} percentile response time while constraining median response time to be no worse than 3500s and 2500s. Observe that as the bound for the median response time becomes tighter (3500s $\rightarrow$ 2500s), OpTune has to trade off a progressively "longer" tail for the desired median performance.

Figure A.3(c) shows that OpTune is also effective when the constraint and optimization goal are exchanged; that is, in these cases, OpTune is seeking to minimize the median performance while observing a constraint for the 99^{th} percentile.

A.3 OpTune Solver Overhead Analysis

Next, we evaluate the performance and overhead of OpTune solver as system scales.

Task	Brute-force	SA
1. min(average)	157s	24s
2. min(median)	163s	32s
3. min(99%ile) and P(50%ile) ≤ 10ms	160s	48s

Table A.3: Running time of different tuning tasks.

A.3.1 Solving Time

We use different tuning tasks and measured the decoding time using both SA and brute-force approach. Table A.3 shows the solving time for different approaches and performance objectives. The time for brute-force is ∼2× higher than SA because brute-force has to explore the entire problem space. As the task varies from single target to multiple targets (e.g., Task 3, details in Section 5.4.3), we observed that the execution time of the SA approach shows a slightly increase, because multi-target performance constraints make SA difficult to find local optimal annealing schedules.

A.3.2 Convergence Speed

Next, we study the solution quality as a function of time. We configure 20 values for the cache parameter and 20 values for the I/O knob in a synthetic two-layer sequential structure. We provide an arbitrary target CDF and perform the decomposition task. We compute the RMSE between the optimal CDF and the target CDF every 10 second. The solution quality is defined $1 - \frac{RMSE}{WorstPerf}$, where the $WorstPerf$ is the maximum performance of the target CDF. Higher value means more close to the target CDF.

Figure A.4 illustrates that SA quickly converges to its optimal compared to brute-force (cut-off at 200 second). We study the convergence speed for both decomposed CDF task (CDF) and multi-point optimization task (multi-points). Interestingly, the brute-force curve is flat and slowly growing because of iterating all cases. The CDF decomposing task using brute-force

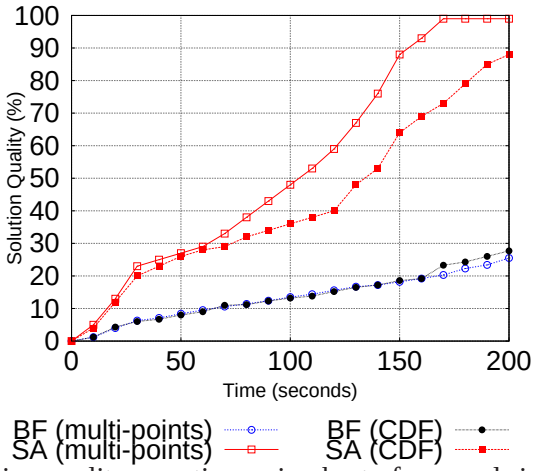


Figure A.4: Solution quality over time using brute-force and simulated annealing.

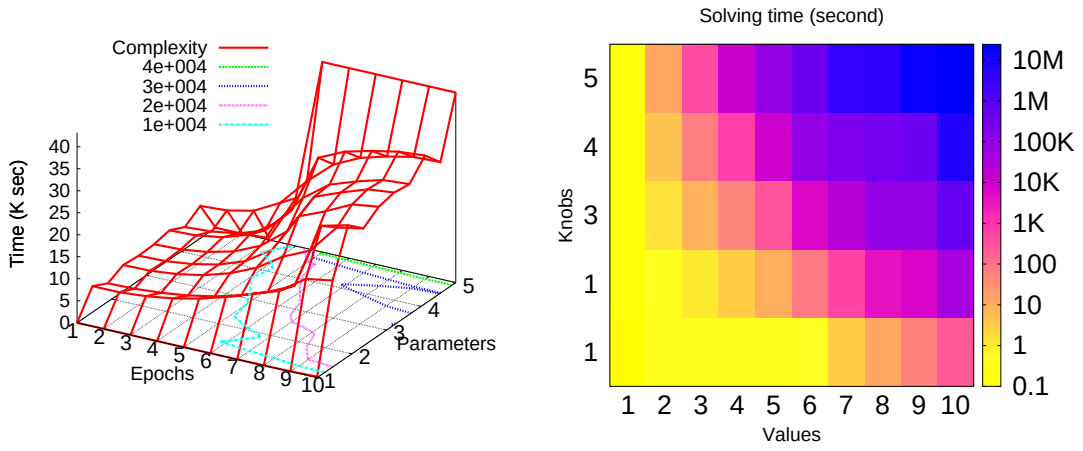
is identical to the multi-points decomposing task curve. We observed that the brute-force approach can only achieve ≤ 23% solution quality at time 200s because it tries every possible solution without memorizing the difference between the current solution and the target solution to decide the next solving schedule. In contrast, the SA curve grows quickly because of reducing impossible solution space. The SA multi-points decomposed curve has faster convergence speed because of the loose constraints compared to SA CDF decompose task.

A.3.3 Extrapolation of Solving Time

Next, we study the solving time as a function of number of configuration parameters and number of epochs for the brute-force approach. In addition to the three systems, we also explore the solver running time in a synthetic multi-tier systems. The experiment simulates a 5-tier system (e.g., TCP/IP stack) and the epoch number is set to 10. Each profiled CDF contains 5000 points. We assume sequential pattern as the structure when connecting each tier.

Figure A.5(a) plots the solving time for the 5-tier synthetic system. The y-axis shows the number of parameters, the y-axis shows the number of epochs and the z-axis shows the solving time in K seconds.

Figure A.5(b) plots the extrapolation of time complexity using randomly generated component CDFs samples. The time complexity of the system suggests that (1) the brute-force



(a) Analytic model for solving time.

(b) Solving time contour.

Figure A.5: Time complexity plot as a function of configuration parameters (5) and epoch granularity (10).

approach is plausible when the number of configuration parameters or the number of epochs is low; (2) the brute-force does not scale up to 6 or more parameter values. We found that finer epoch granularity indicates higher decoding time complexity. The solving time for the brute-force grows rapidly as the system scales.

Note that the each sub-problem in brute-force can be solved independently, therefore the time complexity can be further reduced if we assume a reasonable parallel speedup factor. Another way to reduce time complexity is to provide approximate result. For example, we can return the first result that satisfy the constrains not exploring the entire search space.

References

[1] SWORD: A Developer Toolkit for Web Service Composition. WWW, 2002.

[2] Fusion-IO. http://www.fusionio.com//, 2005.

[3] Memcached, 2014. `http://www.memcached.org`.

[4] Mongoose, 2014. `http://code.google.com/p/mongoose`.

[5] Squid caching proxy, 2014. `http://www.squid-cache.org`.

[6] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE TKDE*, 1999.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. USENIX ATC, 2008.

[8] Apache. Apache Nutch. http://nutch.apache.org/.

[9] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-box Systems. SOSP, 2001.

[10] J. Axboe. CFQ IO Scheduler. Talk presented at linux.conf.au, Jan. 2007.

[11] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. NSDI, 2011.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. ACM SIGCOMM, 2011.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield. Xen 2002. *Technical Report of University of Cambridge*, 2003.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, and R. Neugebauer. Xen and the Art of Virtualization. *SOSP*, 2003.

[15] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *SIGOPS Operating Systems Review*, 2007.

[16] J.-Y. Boudec. Rate adaptation, congestion control and fairness: A tutorial. *EPFL TR*, 2000.

[17] M. Busari and C. Williamson. ProWGen: A Synthetic Workload Generation Tool for Simulation Evaluation of Web Proxy Caches. *Computer Networks*, 2002.

[18] G. C. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[19] P. Cao and S. Irani. Cost-aware WWW Proxy Caching Algorithms. USITS, 1997.

[20] F. Casati, S. Ilnicki, L.-j. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. CAiSE, 2000.

[21] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. ASPLOS, 2009.

[22] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. *USENIX ATC*, 1995.

[23] J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2001.

[24] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. FAST, 2011.

[25] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. MASCOTS, 2011.

[26] L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical Report HPL-98-69R1, Computer Systems Laboratory, HP, 1998.

[27] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. OSDI, pages 217–231, Broomfield, CO, Oct. 2014. USENIX Association.

[28] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[29] C. Constantinescu, J. Glider, and D. Chambliss. Mixing Deduplication and Compression on Active Data Sets. DCC, 2011.

[30] R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4), 2011.

[31] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 2013.

[32] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage deduplication using flash memory. USENIX ATC, 2010.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. SOSP, 2007.

[34] T. Deselaers, S. Hasan, O. Bender, and H. Ney. A deep learning approach to machine transliteration. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 233–241. Association for Computational Linguistics, 2009.

[35] C. Dirik and B. Jacob. The Performance of PC Solid-state Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. *ISCA*, 2009.

[36] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. *SOSP*, 1999.

[37] Facebook Inc. Facebook FlashCache, 2013. `https://github.com/facebook/flashcache`.

[38] J. Feng and J. Schindler. A Deduplication Study for Host-side Caches in Virtualized Data Center Environments. MSST, 2013.

[39] O. Flores and M. Orozco. NucleR: A Package for Non-Parametric Nucleosome Positioning. *Bioinformatics*, 2011.

[40] R. Geambasu, S. D. Gribble, and H. M. Levy. CloudViews: communal data sharing in public clouds. HotCloud, 2009.

[41] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-Resource Fair Queueing for Packet Processing. SIGCOMM, 2012.

[42] P. Gill, M. Arlitt, N. Carlsson, A. Mahanti, and C. Williamson. Characterizing Organizational Use of Web-Based Services: Methodology, Challenges, Observations, and Insights. *ACM Transactions on the Web*, 2011.

[43] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. ASPLOS, 2015.

[44] Google LZ4: Extremely Fast Compression algorithm. Google, 2013. `http://code.google.com/p/lz4`.

[45] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. *OSDI*, 2010.

[46] D. Gupta, L. Checkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. *Middleware*, 2006.

[47] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. ASPLOS, 2015.

[48] L. Huang, G. Peng, and T. Chiueh. Multidimensional Storage Virtualization. *SIGMETRICS*, 2004.

[49] W. Huang et al. A Compression Layer for NAND Type Flash Memory Systems. ICITA, 2005.

[50] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[51] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. SOSP, 2001.

[52] M. Jeon, Y. Kim, J. Hwang, J. Lee, and E. Seo. Workload Characterization and Performance Implications of Large-Scale Blog Servers. *ACM TOW*, 2012.

[53] W. Jin, J. Chase, and J. Kauer. Interposed Proportional Sharing for Storage Service Utility. *SIGMETRICS*, 2004.

[54] W. Josephson, L. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. *FAST*, 2010.

[55] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production Windows Servers. IISWC, 2008.

[56] T. Kgil, D. Roberts, and T. Mudge. Improving NAND Flash Based Disk Caches. *ISCA*, 2008.

[57] Kgil, Taeho and Roberts, David and Mudge, Trevor. Improving NAND Flash Based Disk Caches. ISCA, 2008.

[58] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. FAST, 2008.

[59] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-Aware Virtual Machine Scheduling for I/O Performance. *VEE*, 2009.

[60] J. Kim et al. Deduplication in SSDs: Model and Quantitative Analysis. MSST, 2012.

[61] L. Kish. *Survey Sampling*. 1965.

[62] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. *ISPASS*, 2007.

[63] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM TOS*, 2010.

[64] H. Kushner and P. Whiting. Convergence of Proportional-fair Sharing Algorithms Under General Conditions. *Wireless Communications, IEEE Transactions on*, 2004.

[65] P. Lama and X. Zhou. Efficient Server Provisioning with Control for End-to-End Response Time Guarantee on Multitier Clusters. *IEEE TPDS*, 2012.

[66] S. Lee et al. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM TECS*, 2007.

[67] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. *SIGMOD*, 2008.

[68] C. Li, I. Goiri, A. Bhattacharjee, R. Bianchini, and T. Nguyen. Quantifying and Improving I/O Predictability in Virtualized Systems. IWQoS, 2013.

[69] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim. Assert(!Defined(Sequential I/O)). HotStorage, 2014.

[70] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. USENIX ATC, 2014.

[71] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A Container-based Flash Cache for Compound Objects. ACM/IFIP/USENIX Middleware, 2015.

[72] C. Li, S. Zou, and L. Chu. Online Learning Based Internet Service Fault Diagnosis Using Active Probing. ICNSC, 2009.

[73] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. FAST, 2013.

[74] S. L. Lohr. *Sampling: Design and Analysis*. 2009.

[75] T. Makatos et al. Using Transparent Compression to Improve SSD-based I/O Caches. EuroSys, 2010.

[76] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian. Sar: Ssd assisted restore optimization for deduplication-based storage systems in the cloud. In *IEEE 7th International Conference on Networking, Architecture and Storage (NAS)*, pages 328–337, 2012.

[77] G. Mateescu, W. Gentzsch, and C. J. Ribbens. Hybrid computingwhere hpc meets grid and cloud computing. *Future Generation Computer Systems*, 27(5):440–453, 2011.

[78] J. Matthews et al. Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM TOS*, 2008.

[79] R. McDougall. Filebench: Application Level File System Benchmark. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[80] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. FAST, 2003.

[81] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput Using Solid State Drives (SSD). MSST, 2010.

[82] D. Meyer. Virtual Disk Backend Driver for Xen. `http://wiki.xensource.com/xenwiki/blktap2`.

[83] Micron MLC Flash-based SSD Specification, 2013. `http://www.micron.com/products/nand-flash/`.

[84] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. OSDI, 2014.

[85] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rostron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. *Eurosys*, 2009.

[86] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *EuroSys*, 2010.

[87] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. *USENIX ATC*, 2013.

[88] J. Ouyang et al. SDF: Software-defined Flash for Web-scale Internet Storage Systems. ASPLOS, 2014.

[89] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the ACM European Conference on Computer Systems*, 2007.

[90] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. SIGCOMM, 2012.

[91] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient Guaranteed Disk Request Scheduling with Fahrrad. *Eurosys*, 2008.

[92] T. Prichett and M. Thottethodi. SieveStore: A Highly-Selective, Ensemble-Level Disk Cache for Cost-Performance. *ISCA*, 2010.

[93] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. EuroSys, 2012.

[94] E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *SIGMETRICS*, 1998.

[95] D. Shue and M. J. Freedman. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. OSDI, 2012.

[96] D. Skinner and W. Kramer. Understanding the Causes of Performance Variability in HPC Workloads. IISWC, 2005.

[97] J. Slauson and Q. Wan. Approximate Hadoop, 2012. `http://www.joshslauson.com/pdf/cs736_project.pdf`.

[98] Solomon Hykes. Docker. http://www.docker.com//, 2008.

[99] K. Srinivasan et al. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. FAST, 2012.

[100] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. NSDI, 2005.

[101] Storage Networking Industry Association. MSR Cambridge Traces. `http://iotta.snia.org/traces/158`, 2010.

[102] TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). `http://www.t13.org`, 2007.

[103] H. Tseng, H. Li, and C. Yang. An Energy-Efficient Virtual Memory System with Flash Memory as the Secondary Storage. *ISLPED*, 2006.

[104] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger. Argon: Performance insulation for shared storage servers. *FAST*, 2007.

[105] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. FAST, 2012.

[106] J. Wang, P. Varman, and C. Xie. Avoiding Performance Fluctuation in Cloud Storage. *HiPC*, 2010.

[107] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[108] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of scsi disk drive parameters. *SIGMETRICS*, 1995.

[109] J. Wu and S. Brandt. The Design and Implementation of Aqua: An Adaptive Quality of Service-Aware Object-Based Storage Device. *MSST*, 2006.

[110] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile Main Memory Storage System. *ASPLOS*, 1994.

[111] E. Xun and C. Li. Tree Mapping Template for Prosodic Phrase Boundary Predication. *Journal of Chinese Language and Computing (COLIPS)*, 2007.

[112] E. Xun and C. Li. Applying Terminology Definition Pattern and Multiple Features to Identify Technical New Term and its Definition. *Journal of Computer Research and Development*, 2009.

[113] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. HPCA, 2011.

[114] Y. Yang, M. Dumas, L. García-Bañuelos, A. Polyvyanyy, and L. Zhang. Generalized Aggregate Quality of Service Computation for Composite Services. *Journal of Systems and Software*, 2012.

[115] K. S. Yim, H. Bahn, and K. Koh. A Flash Compression Layer for Smartmedia Card Systems. *IEEE TOCE*, 2004.

[116] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. SIGCOMM, 2012.

[117] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage Performance Virtualization Via Throughput and Latency Control. *ACM TOS*, 2006.

[118] X. Zhang, K. Davis, and S. Jiang. QoS Support for End Users of I/O-intensive Applications using Shared Storage System. *SC*, 2011.

[119] H. Zheng, J. Yang, W. Zhao, and A. Bouguettaya. QoS Analysis for Web Service Compositions Based on Probabilistic QoS. ICSOC, 2011.

[120] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. USENIX, 2009.

[121] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic Configuration of Internet Services. Eurosys, 2007.