3-18-2016

Flash Caching for Cloud Computing Systems

Dulcardo Ariel Arteaga Clavijo
Florida International University, darte003@fiu.edu

DOI: 10.25148/etd.FIDC000230

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

FLASH CACHING FOR CLOUD COMPUTING SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Dulcardo A. Arteaga Clavijo

2016

To: Interim Dean Ranu Jung
College of Engineering and Computing

This dissertation, written by Dulcardo A. Arteaga Clavijo, and entitled Flash Caching for Cloud Computing Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Raju Rangaswami

Jason Liu

Gang Quan

Swaminathan Sundararaman

Ming Zhao, Major Professor

Date of Defense: March 18, 2016

The dissertation of Dulcardo A. Arteaga Clavijo is approved.

Interim Dean Ranu Jung
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and
Economic Development and Dean of the
University Graduate School

Florida International University, 2016

DEDICATION

To my beloved parents.

ACKNOWLEDGMENTS

Foremost I would like to express my deepest gratitude to my advisor, Professor Ming Zhao, for his support and guidance throughout my PhD study. He deserves special thanks for his patience in teaching a nonnative speaker to write and speak in English. I would also like to thank his wife, Dr. Jing Xu, for helping me to apply forecasting techniques to on-demand cache allocation.

I want to thank my committee members—Professor Raju Rangaswami, Professor Jason Liu, Professor Gang Quan, and Dr. Swaminathan Sundararaman–for reviewing my proposal and dissertation and for offering helpful comments to improve my work.

I also wish to thank all members of the VISA lab–Lixi Wang, Xiqi Xu, Douglas Otstoot, Jorge Cabrera, Greg Jean-Baptise, Wenji Li, and Saman Biook–for their help with proofreading, rehearsing, and providing feedback on my papers and dissertation. Special thanks to Jorge Cabrera who helped me to improve our simulator and conduct experiments.

ABSTRACT OF THE DISSERTATION

FLASH CACHING FOR CLOUD COMPUTING SYSTEMS

by

Dulcardo A. Arteaga Clavijo

Florida International University, 2016

Miami, Florida

Professor Ming Zhao, Major Professor

As the size of cloud systems and the number of hosted virtual machines (VMs) rapidly grow, the scalability of shared VM storage systems becomes a serious issue. Client-side flash-based caching has the potential to improve the performance of cloud VM storage by employing flash storage on VM hosts to exploit the locality inherent in VM I/Os. However, there are several challenges to the effective use of flash caching in cloud systems. First, cache configurations such as size, write policy, metadata persistency, and redundant array of independent disks (RAID) level have a significant impact on flash caching. Second, the typical capacity of flash devices is limited in comparison to the dataset size of consolidated VMs. Finally, flash devices wear out and face serious endurance issues that are aggravated by the use of caching.

This dissertation presents research on how to address problems of cloud flash caching in the following three aspects: First, it presents a thorough study of different cache configurations, including a new cache-optimized RAID configuration that uses a large number of long-term traces collected from real-world public and private clouds. Second, it studies an on-demand flash cache management solution for meeting VM cache demands and minimizing device wear out. It uses a new cache demand model, reuse working set (RWS), to capture data with good temporal locality, and uses RWS size (RWSS) to model a workloads cache demand. Finally, in situations where a cache is insufficient to meet VM demand, it employs dynamic

cache migration to balance cache load across hosts by live-migrating cached data along with the VMs.

The results show that, compared to traditional RAID, cache-optimized RAID improves performance by 137% without sacrificing reliability. In addition, compared to traditional working-set-based cache allocation, RWSS-based on-demand cache allocation reduces workload cache usage by 78% and lowers the amount of writes sent to the cache device by 40%. Combining on-demand cache allocation with dynamic cache migration for 12 concurrent VMs yields a 28% higher hit ratio and 28% lower 90th percentile I/O latency, compared to cases without cache allocation.

TABLE OF CONTENTS

CHAPTER PAGE

1. Introduction 1
1.1 Thesis Statement 3
1.2 Contributions 4
1.3 Outline 5

2. Flash Caching Architecture and Configurations 6
2.1 Introduction 6
2.2 Background and Motivation 9
2.3 Methodology 11
2.3.1 Dm-cache Block-level Cache 11
2.3.2 Dm-cache-sim Cache Simulator 13
2.3.3 Traces 13
2.3.4 Experimental Testbed 15
2.4 Cacheability Analysis 17
2.5 Cache Overhead 19
2.6 Latency Analysis 23
2.6.1 FIO Benchmark 23
2.7 Write Policy Analysis 27
2.7.1 IO Latency 28
2.7.2 Server Load 32
2.8 Persistency Analysis 34
2.8.1 Persistency Overhead 34
2.8.2 Persistency Benefits 37
2.9 Reliability 39
2.10 Summary 42

3. On-demand Space Allocation 44
3.1 Introduction 44
3.2 Motivations 46
3.3 Architecture 48
3.4 On-demand Cache Allocation 50
3.4.1 RWS-based Cache Demand Model 50
3.4.2 Online Cache Demand Prediction 54
3.4.3 Cache Allocation and Admission 55
3.4.4 Evaluation 57
3.5 Summary 66

4. Dynamic Cache Migration 67
4.1 Introduction 67
4.2 Live Cache Migration 68
4.3 Migration Rate Limiting 71
4.4 Evaluation 72
4.5 Putting Everything Together 75
4.6 Summary 78

5. Related Work 81
5.1 Flash Caching Solution 81
5.2 Cache Management 83
5.3 Cache Migration 85

6. Conclusions and Future work 86
6.1 Conclusions 86
6.2 Future Work 87

BIBLIOGRAPHY 89

VITA 95

LIST OF FIGURES

FIGURE PAGE

2.1 Architecture of Shared flash Caches for Cloud 12

2.2 FIU Web server IO patterns 15

2.3 FIU Moodle server IO patterns 15

2.4 FIU Buffalo file server IO patterns 16

2.5 FIU Bear file server IO patterns 16

2.6 Cloud VPS VM IO patterns 17

2.7 Working set size (WSS) variations over time 19

2.8 Cache hit rate given different cache Size and WSS 20

2.9 Dm-cache latency for read workloads 21

2.10 Dm-cache latency for write workloads 21

2.11 Sequential Reads 24

2.12 Sequential Writes 25

2.13 Sequential Reads-Writes 26

2.14 Random Reads 27

2.15 Random Writes 28

2.16 Random Reads-Writes 29

2.17 Performance of a read-intensive workload using different write caching policies 30

2.18 Performance of a write-Intensive workload using different write caching policies 30

2.19 Server IO load for Web server trace 33

2.20 Server IO load for Moodle server trace 33

2.21 Server IO load for Buffalo server trace 34

2.22 Server IO load for Bear server trace 34

2.23 Server IO load for CloudVPS traces 35

2.24 Persistency overhead with fio 37
2.25 Cache hit rate changes over time with different persistency configurations 38
2.26 Overhead of cache-optimized RAID 39
2.27 Performance of different reliability configurations 40
3.1 Architecture of CloudCache 48
3.2 RWS analysis using different values of N 52
3.3 Time window analysis for the Moodle trace 54
3.4 RWSS-based cache demand prediction 56
3.5 Prediction accuracy 59
3.6 Staging strategy analysis 61
3.7 Comparison to HEC 62
3.8 Allocation methods 63
3.9 VM IO latency comparison 64
4.1 Architecture of dynamic cache migration 70
4.2 Migration strategies 73
4.3 Impact of different cache migration rate 75
4.4 Cache usages of 12 concurrent VMs 80

CHAPTER 1

Introduction

Network storage systems such as SAN [TS00] and IP-SAN (e.g., iSCSI [KHSB02], NBD [nbd]) are commonly used in emerging cloud computing systems to store virtual machine (VM) images for a set of VM hosts (e.g., [ebs, nov]). Such a shared storage system allows efficient storage utilization by consolidating separate VM storage resources into a single shared pool. It also enables fast, live VM migrations, which transfer only the VMs' in-memory states across hosts during migrations. However, as the size of cloud systems and the number of hosted VMs rapidly grow, the scalability of shared storage becomes a serious issue. In production cloud deployment, a single host can run hundreds of VMs whereas a cluster of hosts can run thousands of VMs sharing the same storage server. Consequently, the VM storage system may become the bottleneck where VMs cannot reach their desired performance even when provisioned with the necessary CPUs and memory.

Cloud system host-side *flash caching* employs flash-memory-based storage on a VM host as the cache for its remote storage to exploit the data-access locality and improve VM performance. It has received much attention in recent years [BLM+12, ioC, HAWS13, dmc], for two important reasons. First, as the level of consolidation continues to grow in cloud computing systems, the scalability of shared VM storage servers becomes a serious issue. Second, the emergence of flash-memory-based storage has made flash caching a promising option to address the issue of I/O scalability because accessing a local flash cache is significantly faster than accessing remote storage across a network.

However, several challenges must be addressed to make effective use of flash caches in cloud storage systems:

Flash caching architecture and configurations First, it is important to properly configure the cache to deliver the best possible performance with data reliability. To do so, several key questions must be answered. First, *how to size the flash caches?* Given the capacity and cost constraints of flash devices, there needs to be enough locality in VM IOs in order to make flash caching cost-effective. Otherwise, the cloud may not be a good target for flash caching. Second, *how to choose the write caching policies?* Although the nonvolatile nature of flash storage allows writes to be served directly from the cache, synchronizing the cache with the server has implications for both IO performance and data durability. Third, *is it necessary to make a flash cache persistent across client restarts and crashes?* A persistent cache requires both data and metadata to be persistently stored in the cache, which introduces additional writes that are detrimental to both IO performance and flash endurance. Finally, *how to improve the reliability of a flash cache so as to tolerate device-level failures?* If a flash cache retains locally modified data, it is critical that it can recover from flash device failures, but the fault-tolerance mechanism employed should not negate the performance benefits of write-back caching.

Limited cache capacity Second, cache capacity is very limited in comparison to VM dataset sizes. Considering the increasing data intensity of workloads and the increasing number of VMs consolidated to a host, it is important to allocate shared cache capacity among competing VMs according to their actual demands.

Limited device endurance Third, flash devices wear out by writes and face serious endurance issues, which are in fact aggravated by the use for caching because both the writes inherent in the workload and the reads that miss the cache induce wear-out [YPGT13]. Therefore, cache management must be

careful not to admit data that are not useful to workload performance but damage the endurance.

We believe that these problems are significant and that solving them would support the effective use of cloud caching to address the issue of scalability on cloud storage and optimize the performance of cloud applications. This thesis presents a flash-caching solution that first considers a feasibility study of real cloud-system traces then proposes a solution that considers on-demand cache allocation of cache capacity according to VM workload demands, and dynamic cache migration that balances cache loads across hosts by live-migrating VMs along with their cached data.

1.1 Thesis Statement

We propose "CloudCache", a cloud caching and management solution, that addresses the previous challenges, in the following ways:

1. Provides a thorough study of cache configurations such as size, write policy, metadata persistency, and reliability, as well as their impact on cloud caching performance and reliability
2. Utilizes on-demand space allocation, which uses reuse working set size (RWSS) model to predict workload demand, to effectively admit reused data into the cache and enforce cache allocation
3. Employs dynamic cache migration which allows to migrate cached data across hosts along with the VM, in order to maintain cache load balance across multiple hosts.

1.2 Contributions

The first contribution is a comprehensive study of different cache configurations and their impact on performance and reliability. First, by comparing the working set size (WSS) of the traces to the typical size of commodity flash devices, we validate that cloud systems are strong targets for flash caching. Second, by studying different cache write policies, we determine the tradeoff of each. Third, we study the overheads involved in making cache metadata persistent. Finally, with an understanding of the importance of write-back caching, we investigate how to make the process reliable and affordable. Our solution is a new cache-optimized RAID technique that selectively provides data redundancy.

The second contribution is on-demand cache-space allocation, which enhances our caching solution with the capacity to analyze current workload demand by using the RWS model. Based on this model, we study prediction methods to estimate a workloads cache demand based on the observed RWSS and cache admission policies to admit only data with good temporal locality, thereby maximizing workload performance while minimizing wear. Our solution is then able to allocate shared cache capacity to the VMs according to their actual cache demands.

The third contribution is dynamic cache migration. Because it is important to maintain cache load balance across hosts in order to satisfy VM cache demands, dynamic cache migration approach helps to balance cache load across hosts by live-migrating cached data along with the VMs. It uses both on-demand migration of dirty data to minimize the overhead of synchronizing such data and background migration of reuse working set to quickly warm up the cache for a migrated VM, thereby minimizing impact on the VM. Meanwhile, it can also limit the data transfer rate for cache migration to minimize the impact on cohosted VMs.

Our results show that, compared to traditional RAID, cache-optimized RAID improves performance by 137% without sacrificing reliability. In addition, compared to traditional working-set-based cache allocation, RWSS-based on-demand cache allocation reduces workload cache usage by 78% and lowers the number of writes sent to cache device by 40%. Combining on-demand cache allocation with dynamic cache migration for 12 concurrent VMs yields a 28% higher hit ratio and a 28% lower 90th percentile I/O latency, compared to cases without cache allocation.

1.3 Outline

The rest of the thesis is organized as follow: Chapter 2 presents the study on different cache configurations using real cloud traces; Chapter 3 presents on-demand space allocation which allocates cache capacity according to VMs' workload demands; Chapter 4 presents dynamic cache migration which is used to balance cache load across hosts; Chapter 5 describes the related work; and finally, Chapter 6 presents the thesis conclusions and future work.

CHAPTER 2

Flash Caching Architecture and Configurations

2.1 Introduction

Flash caching has become a popular and effective solution for improving I/O performance for computing systems that depend on network storage system for example data centers and cloud providers, which can host tens to hundreds of VMs in the same physical hosts.

However, several key questions need to be answered to make effective use of flash caches in cloud storage systems. First, *how to size the flash caches?* Given the capacity and cost constraints of flash devices, there needs to be enough locality in VM IOs in order to make flash caching cost effective. Otherwise, cloud may not be a good target for flash caching. Second, *how to choose the write caching polices?* Although the non-volatile nature of flash storage allows writes to be served directly from the cache, how to synchronize the cache with the server has implications on both IO performance and data durability. Third, *is it necessary to make a flash cache persistent across client restarts and crashes?* A persistent cache requires both data and metadata to be persistently stored in the cache, introducing additional writes, which are detrimental to both IO performance and flash endurance. Finally, *how to improve the reliability of a flash cache so as to tolerate device-level failures?* If a flash cache retains locally modified data, it is critical that the cache can recover from flash device failures, but the fault-tolerance mechanism employed should not negate the performance benefits of write-back caching.

This chapter studies client-side flash caching in cloud systems by providing answers to the above questions based on *dm-cache* [dmc], a block-level cache solution

that provides transparent flash caching on cloud VM hosts and supports concurrent, dynamic VMs to efficiently share a cache. It has been adopted by cloud service providers for production use [cloa]. To facilitate this study, we have collected a substantial amount of block IO traces from a private cloud at Florida International University (FIU) and a public cloud from CloudVPS [cloa]. The FIU traces contain nearly one year of block IO traces collected from several production servers (Web serve, Moodle server, and network file system servers). The CloudVPS traces contain block IO traces from hundreds of VMs on the production systems of the Infrastructure-as-a-Service (IaaS) cloud for several days.

Our study first analyzes the basic characteristics of dm-cache based flash caching and the collected cloud traces. It reveals that dm-cache introduces small latency overhead which is around 23μs when using an SATA solid-state drive (SSD) devices and 9μs when using a PCIe SSD device. It also validates that cloud VMs are good targets for flash caching by comparing the working set size (WSS) of the traces to the typical size of commodity flash devices.

Having confirmed the feasibility of flash caching, we further study the impact of different write caching policies. Our results show that retaining writes in the cache is beneficial to performance, producing a 48% to 321% speedup in comparison to a policy that only invalidate cache blocks upon writes. More importantly, delaying the synchronization with the server (i.e., write-back caching) can significantly improve the performance, producing 74% to 1289% speedup compared to the policy that synchronizes with the server upon every write (i.e., write-through caching). This improvement is mainly attributable to a 52% to 94% reduction of server load by exploiting the locality of cached writes, which was not considered in related studies [HAWS13]. Our study also reveals the tradeoff associated with making a flash cache persistent across client restarts and crashes. To store the metadata persis-

tently, it introduces up to 0.06ms latency overhead, but it allows the client to work with a warm cache after it recovers, which saves the time (3 to 5 hours in our traces) to warm up the cache and increases the hit rate by up to 28%. Compared to the related work [HAWS13], we provide quantitative results on the cost and benefit of making flash cache persistent and show that this tradeoff should be carefully decided based on the cloud environment such as the expected client failure rate.

Understanding the importance of write-back caching, we further investigate how to make it reliable and affordable. Our solution is a new cache-optimized RAID technique that selectively provides data redundancy. It recognizes the fact that cached clean data already have redundant copies on the server and employs additional flash devices only to provide fault tolerance to cached dirty data, thereby minimizing the overhead while maximizing utilization. The results show that this cache-optimized RAID can provide fault tolerance with negligible overhead (9.1μs), and substantially improves the performance by 135% and 72% compared to using traditional RAID and write-through caching, respectively, to achieve reliability.

Overall the work described in this chapter has made the following contributions: *1)* it provides one of the first comprehensive analysis of the effectiveness of flash caching based on a production-grade cache utility designed for cloud environments and a substantial amount of traces collected from real-world systems; *2)* It is among the first to quantitatively analyze the various key cache design decisions for flash caching, and demonstrate the significance of employing write-back policy and making a flash cache persistent across client restarts; *3)* It proposes a new cache-optimized RAID technique to allow a write-back cache to tolerate flash device failures with minimal cost.

The rest of the chapter is organized as follow: Section 2.2 describes the background and related work; Section 2.3 presents the methodology of the trace-driven

analysis; Section 2.4 analyzes the cacheability of cloud workloads using the collected cloud traces; Section 2.5 discusses the overhead of dm-cache-based flash caching; Section 2.6 shows a latency analysis; Section 2.7 analyzes the impact of different write caching policies; Section 2.8 analyzes the cost and benefit associated with making a cache persistent; Section 2.9 presents the cache-optimized RAID technique for reliable write-back caching; and Section 2.10 summarizes the chapter.

2.2 Background and Motivation

Client-side persistent-storage-based caching can improve the performance of a distributed storage system by harnessing the persistent storage available on the storage client to exploit the locality within its IOs, thereby accelerating data accesses to the client and reducing IO load on the server. Earlier results from dm-cache show that HDD-based client-side caching can achieve a 15-fold speedup for an iSCSI-based system with 8 clients sharing one HDD-based server [HZ06]. However, the use of client-side disk caching was not widely adopted, which can be attributed to at least the fact that the latency of an HDD-based cache is often comparable to the network latency to the storage server. Therefore, the benefit of client-side caching hence exhibits only when the server is heavily loaded [HZ06] or accessed through a wide-area network [ZF06, ZZF06].

While the emergence of flash-based storage is fundamentally transforming the landscape of computer storage field, it is also changing the perception on client-side caching, because the speed of a flash-based cache can be substantially faster than an HDD-based storage server. Even as flash storage gets increasingly adopted on the storage server side, the diversity of flash devices allows the use of faster flash

storage (e.g., single-level cell flash) on the client side as the cache for the slower flash storage (e.g., multiple-level cell flash, hybrid flash/HDD) on the server side.

The potential of flash caching has motivated several SSD caching solutions (e.g., dm-cache [dmc], ioCache [ioC], Mercury [BLM+12]). In this chapter, we focus on *dm-cache* based SSD caching, which has been successfully deployed in production cloud computing systems and motivated the designs of other SSD caching solutions (e.g., FlashCache [fla]).

Although the potential of client-side flash caching is well recognized, it is still unclear how much performance improvement that it can achieve for typical cloud workloads and how to best design and configure the cache given the many possible choices. Recently, Holland et al. [HAWS13] studied several key design considerations, including flash-RAM integration, write-back policy, cache persistency, and cache consistency, based on simulations. In particular, they found that write-through caching is good enough because the writes to the storage server can be submitted asynchronously without slowing down the client. However, they did not consider the impact on the server's load and its resulting effect on the client's performance, which are studied in this chapter using a real flash cache implementation with real traces.

The importance of flash caching has also motivated related work on exploiting cache-specific characteristics to optimize the use of flash storage. FlashTier [SSZ12] studied a new flash device interface specialized for caching, which reduces the block management overhead by unifying the block address mappings done by the cache and device and reduces the device garbage collection overhead by silently evicting clean cache blocks.

HEC [YPGT13] and LARC [HWC+13] described on section 5.1 studied new cache admission policies to address the flash wear-out issue by not caching data that

are infrequently used or from backup workloads. These solutions are complementary to this chapter's study which focuses on the design issues internal to cache while our discoveries also have an impact on flash performance and endurance.

2.3 Methodology

2.3.1 Dm-cache Block-level Cache

Dm-cache [dmc, HZ06] provides caching at block level for distributed storage systems. It is created upon block-level storage virtualization by interposing a virtual block device between the storage client and server, and can be transparently deployed on the client-side of a distributed storage system to provide caching. Our current implementation of dm-cache is based on the Linux block device virtualization framework (device mapper) and can be seamlessly employed by any Linux-based environments including VM systems that use Linux-based IO stack (e.g., Xen [BDF+03] and KVM [kvm]). It is an open-source solution and has been adopted by production cloud systems [clob]. Therefore, we use dm-cache as a representative flash caching solution and feed it with real-world traces to carry out this study.

Dm-cache supports full associativity with LRU-based replacement and various write caching polices. Dm-cache employs a radix tree for fast cache lookup and an LRU list for quickly finding a replacement block. (See Section 2.5 for the overhead analysis.) Although alternative cache replacement algorithms (e.g., ARC [MM03]) are available, cache replacement is not the focus of this chapter and the use of LRU in our study offers at least a baseline performance from a commonly used algorithm.

To support the use in cloud computing systems, dm-cache allows multiple co-hosted VMs to safely share the same cache device in a work-conserving manner

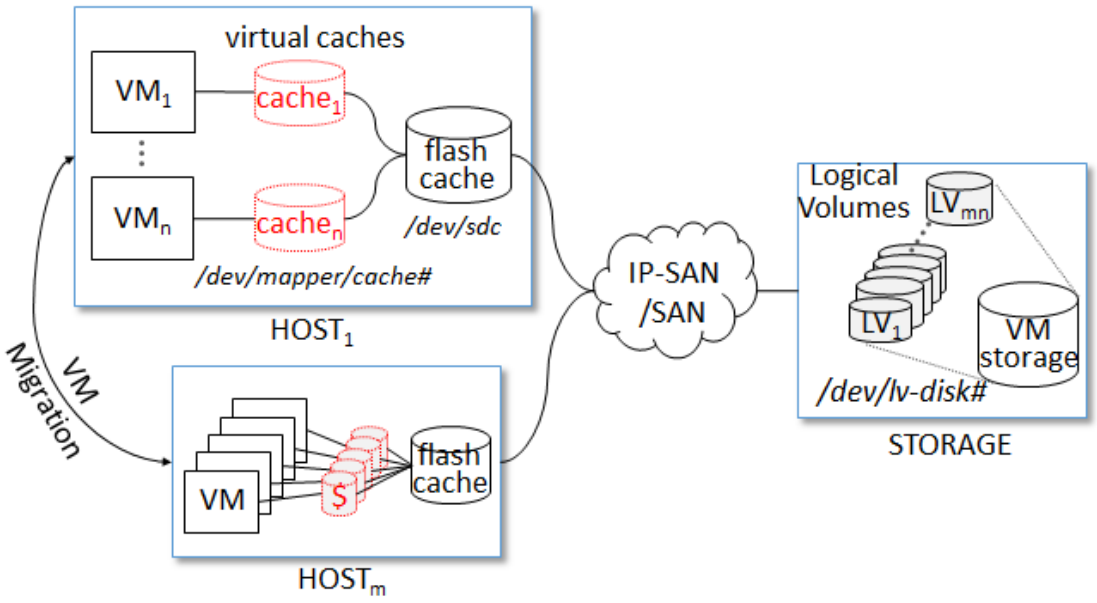


Figure 2.1: Architecture of Shared flash Caches for Cloud

(but with isolated data-sets, data sharing at block level requires a cluster file system [VMF, glo] and will be considered in our future work). It also allows the cache contents to be controlled on a per-VM basis in order to support dynamic VM life cycles and migrations. For example, when a VM is terminated or migrated to a different host, its cached data can be flushed without affecting the other VMs sharing the same cache.

Figure 1 illustrates the architecture of dm-cache-based flash caching in cloud environments. In this example there are multiple VMs each with its own virtual disk stored directly on the logical volumes (LVs) (*/dev/lv-disk#*) remotely accessed through SAN or IP SAN (e.g., iSCSI [KHSB02], NBD [nbd]). The local storage device (*/dev/sdc*) on the client-side of this distributed storage system, the VM host, is used to provide block-level caching for the VM images. In order for the VMs to share the cache device, a virtual cache (e.g., */dev/mapper/cache1*) is created for

each VM and presented to the VM as its virtual disk, while all the virtual caches are at the end mapped to the same physical cache device (a VM identifier is stored in every cached block to track the ownership of cache data). Each VM's IOs to its virtual disk are thereby handled by dm-cache and satisfied from the cache or remote LV.

2.3.2 Dm-cache-sim Cache Simulator

To facilitate the analysis of cache performance using long-term traces and with different cache configurations, we also created a user-level cache simulator, *dm-cache-sim*. It is able to flexibly model the cache management of dm-cache, use block-level traces to drive the simulations, and collect detailed statistics on cache usages and hit rates. Note that we do not use this simulator to gather IO latency or throughput which are always collected using dm-cache with real experiments. Because we do not attempt to simulate the time behavior of dm-cache, the simulator generally runs faster than real experiments while giving the same cache hit rate results as real experiments. It is therefore good for quickly exploring the impact of different cache configurations on cache hit rate using long-term traces.

2.3.3 Traces

To support this flash caching study, real-world block IO traces were collected from production cloud systems using *blktrace*, a Linux block-layer IO tracing mechanism [blk], and, *dtrace* a Solaris dynamic tracing framework [dtr]. The statistics of the collected traces are summarized in Table 3.1. The first group of traces were collected from a private cloud at FIU. Several production servers (Web, Moodle, and network file system servers) were traced for months. The *Web server* hosts a departmental

Server Name	Time (days)	IO Load (GB)	WSS (GB)	Write (%)
webserver	281	2,247	110	51
moodle	161	17,364	223	13
buffalo	90	39,128	638	41
bear	152	57,887	1037	22
CloudVPS (170+ VMs)	3	7 - 223	5 - 20	14 - 85

Table 2.1: Trace statistics

website; the *Moodle* server hosts the Moodle online learning system; the *Bear* and *Buffalo* servers are the file servers for storing the user data of faculty and students, respectively. These different types of servers represent services that are commonly hosted on cloud VMs. The second group of traces were collected from the production system of a public IaaS cloud provider (*CloudVPS*) [cloa]. A random set of 170 VMs were selected from three VM hosts and traced for up to three days.

Figure 2.2 shows the total number of IOs and the numbers of reads and writes for the *Webserver* trace for over 10 months. This workload is write-intensive because the reads to the commonly visited web pages can be well captured by the webserver's memory cache, leaving the underlying storage layer a high ratio of writes. Figure 2.3 shows the patterns of the *Moodle* trace for nearly six months. Although this trace is collected also from a website, its patterns are quite different from the *Webserver*. First, the overall intensity is an order of magnitude higher than the *Webserver* trace, because the Moodle website services contents such as course slides and assignments which are much larger than the data served by the *Webserver*. Second, because the working set is much larger, a significant number of reads misses the memory cache and dominates the storage workload (82% overall). Figures 2.4 and 2.5 show the IO patterns for the two file server traces, which are both much more intensive than the *Webserver* and *Moodle* traces. Between these two file servers, *Bear* services a larger dataset and its storage workload has a greater percentage of reads than *Buffalo*.

The above four traces provide a good representation of cloud workloads with different levels of IO intensity and different read/write ratio. Figure 2.6 further illustrates commercial cloud workload patterns using a subset of the VM traces collected from *Cloud VPS*, where every group of bars corresponds to a one-day trace from one of the VMs. These VMs exhibit diverse IO characteristics in terms of intensity and read/write ratio. As Cloud VPS is an IaaS provider, the guest systems of the VMs are owned by the users and their behaviors can be only observed from outside of the VMs.

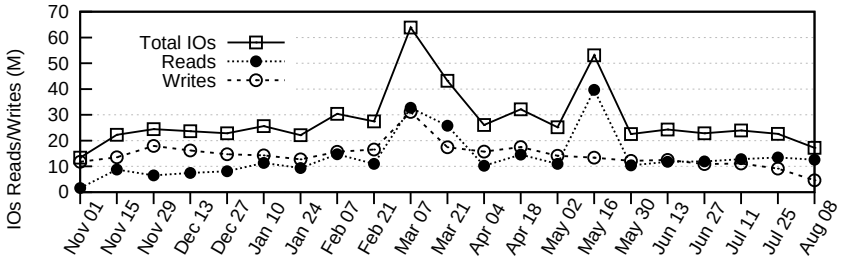


Figure 2.2: FIU Web server IO patterns

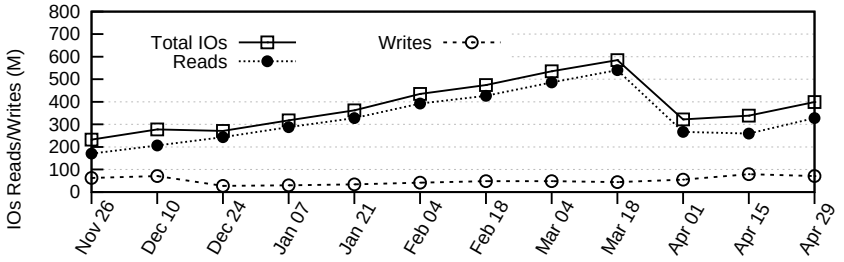


Figure 2.3: FIU Moodle server IO patterns

2.3.4 Experimental Testbed

To obtain IO performance metrics such as latency and throughput of flash caching, the collected traces were replayed on a real iSCSI-based storage system. One node

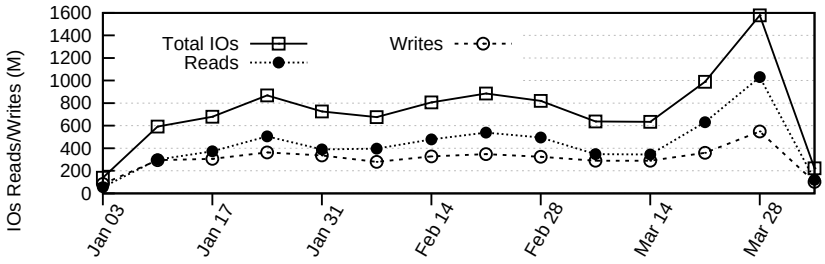


Figure 2.4: FIU Buffalo file server IO patterns

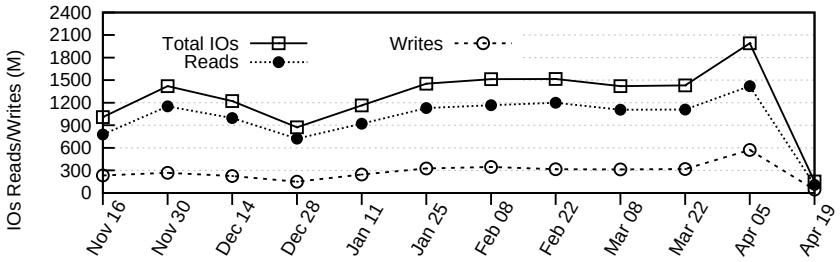


Figure 2.5: FIU Bear file server IO patterns

from a compute cluster is set up as the storage server (iSCSI target) and the others as the clients (iSCSI initiators). Each node has two six-core 2.4GHz Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk, running 3.2.20 Linux-kernel in a Debian 6.0 OS. Each client node in addition is equipped with dm-cache and flash devices to provide caching. The server node runs iSCSI server to export the LVs stored on its SAS disk to the clients via a Gigabit Ethernet.

The performance of flash devices varies across different interfaces, vendors, and models. In this study, we consider two representative devices from major vendors and with different interfaces: a 120GB MLC SATA-interfaced flash device from Intel (Model: Intel C2CW120A3) and a 240GB MLC PCIe interfaced flash device from OCZ.

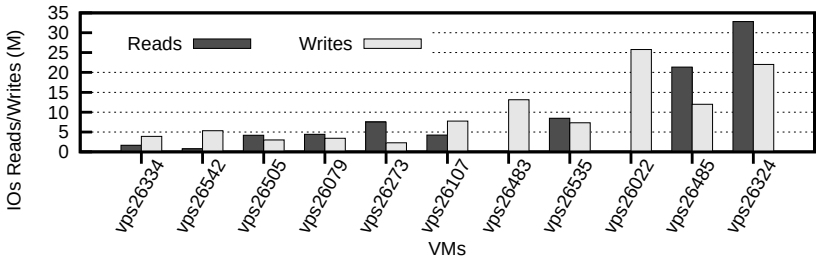


Figure 2.6: Cloud VPS VM IO patterns

Workload type	SATA-SSD (ms)	PCIe-SSD (ms)
Sequential read	0.14	0.23
Sequential write	0.07	0.03
Sequential read/write	0.08	0.16
Random read	0.18	0.23
Random write	0.07	0.04
Random read/write	0.10	0.19

Table 2.2: Raw Flash Latencies

2.4 Cacheability Analysis

We start our study with a basic cacheability analysis by analyzing the working set size (WSS) of our collected cloud traces. Because a cache's performance for a given workload is largely determined by how well the cache can store the workload's working set, we try to understand whether the capacity of commodity flash devices is sufficient with respect to the working set size of a typical cloud workload. In the analysis below, we consider using the write-back caching policy and LRU-based cache replacement.

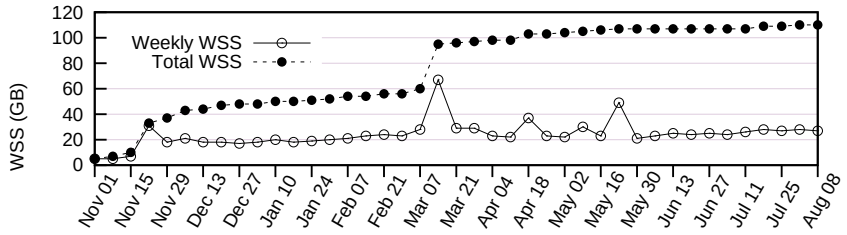
Table 3.1 lists the total WSS, i.e., the total number of unique block references, across the entire duration of every cloud trace. For the Web server, Moodle, and CloudVPS traces, their WSSes can be well stored by a typical commodity flash device. While the WSSes of the buffalo and bear file server traces are much larger,

they can also be completely stored in a high-end flash device. However, in a cloud environment, the limited cache capacity has to be shared by many VMs hosted on the same client. Each VM only gets a portion of the flash cache, which is most unlikely to be sufficient for the total WSSes observed from these traces. Nonetheless, it is also unnecessary to keep the working set of an entire workload which lasts up to 9 months for the above traces, in the cache. If the cache can hold the working set observed at a smaller timer scale, say weeks, then it can still achieve good performance most of the time, except for when the workload transits from one working set to another.

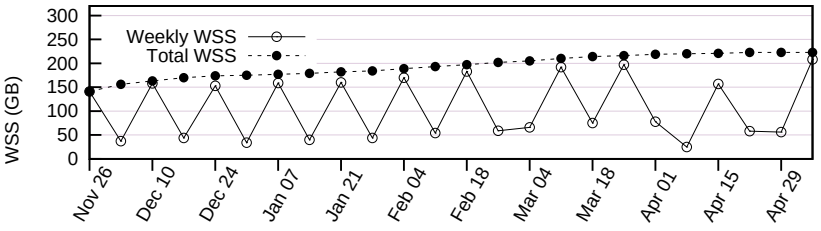
Figure 2.7 shows the WSS calculated per week (*Weekly WSS*) and the WSS calculated from the start of the trace (*Total WSS*) as they vary over time for the Web and Moodle server traces. The results show that the weekly WSS of the Web server workload is quite stable and stays below 20GB most of the time, although the entire WSS for 9 months can grow to 110GB. The weekly WSS of the Moodle server workload fluctuates over time but in average it is 100GB, which is less than half of the total WSS at the end of the 5-month trace.

Finally, to illustrate the potential cache performance with different cache sizes, we extract one-month-long segments of the Web server trace that exhibit different WSS, and show how well the cache performs in terms of hit rate in Figure 2.8. In general, the cache hit rate is well above 50% and exceeds 90% in many cases.

The above analysis reveals that the working sets of typical cloud workloads can be well cached in commodity flash devices, thereby verifying the feasibility of using flash devices as caches in cloud systems. Given the workloads that a VM host need to serve, the above analysis can also help determine the appropriate size of the flash cache. However, for unknown workloads, their WSSes have be estimated online, which will be studied in our future work.



(a) Web Server



(b) Moodle server

Figure 2.7: Working set size (WSS) variations over time

2.5 Cache Overhead

To further investigate the feasibility of flash-based caching, we analyze its worst-case overhead using dm-cache. Cache overhead is directly associated with cache management operations including lookup, insertion, invalidation, and update. This overhead needs to be small, especially considering the fast speed of flash storage which can make any software-introduced overhead appear significant in the overall IO latency. We compare the IO latencies from when flash caching is not used to when it is used but with a cold cache, in order to evaluate the overhead of cache lookup and insertion. We compare the IO latencies from raw flash device (without using dm-cache) to the latencies from warm flash cache (using dm-cache) to evaluate the overhead of cache lookup and invalidation/update.

We use *fio* [fio] to create basic read and write intensive workloads of sequential and random patterns. These workloads are issued to the raw storage device using

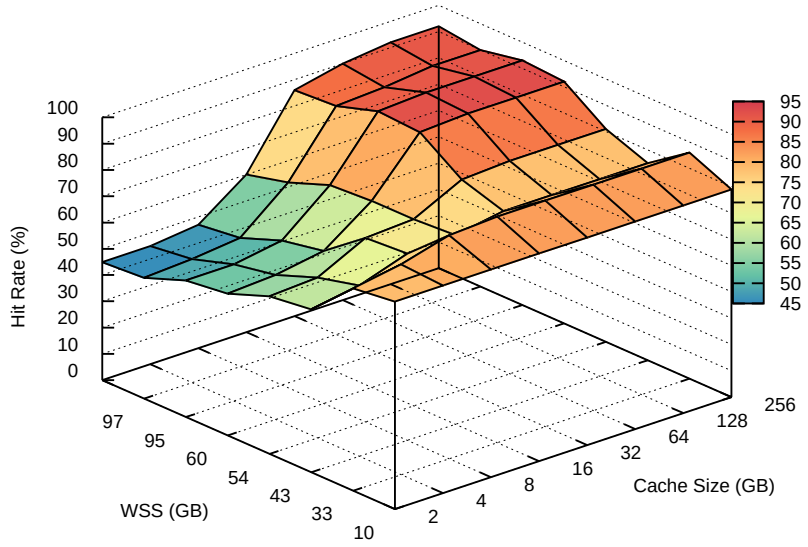


Figure 2.8: Cache hit rate given different cache Size and WSS

direct IOs so that any potential optimization done by the file system and memory cache is bypassed in this performance analysis. Each workload exercises 1GB of data and is repeated four times. We consider three caching policies as defined below, which mainly differ in how they handle a write to the cache.

- *Write-invalidate (WI)*: The write invalidates the cached block and is submitted to the storage server.
- *Write-through (WT)*: The write updates both the cache and the storage server.
- *Write-back (WB)*: The write is stored in the cache immediately but is submitted to the storage server later. Before the storage server gets the write, the block is locally modified in the cache and considered *dirty*.

The write-through and write-back polices are well studied in the processor cache related literature [HP06], which provide a tradeoff between data coherence and performance. The write-invalidate policy sounds contrived, but it simplifies the han-

dling of writes. There are also some variations in implementing the write-through and write-back policies. For write-through, the IO experiences a write stall if it waits for the write to complete in both the cache and back-end storage [HP06]. A common optimization is to allow the IO to be returned to the upper storage layer once it is stored in the cache, which allows the application to continue while the back-end storage is being updated. The dm-cache implementation adopts this optimization. For the write-back policy, a dirty block is written to the back-end storage when it is replaced. As an optimization for better data reliability, dm-cache also supports the automatic flushing of dirty blocks periodically or when the percentage of dirty blocks exceeds a threshold (similarly to the Linux *pdflush* policy) as well as manual flushing through a IOCTL signal.

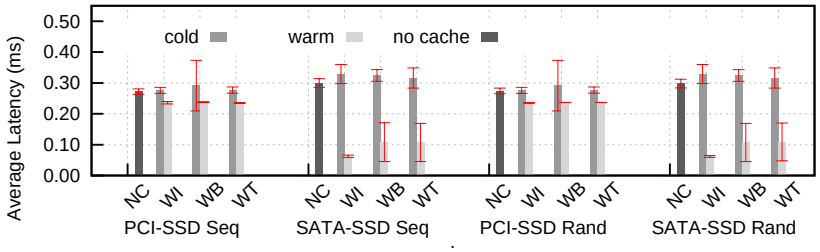


Figure 2.9: Dm-cache latency for read workloads

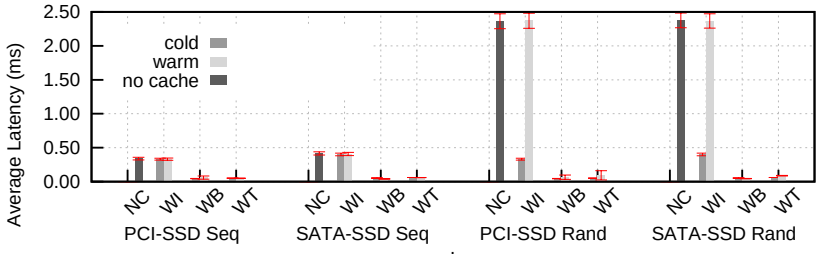


Figure 2.10: Dm-cache latency for write workloads

For a read workload (Figure 2.9), the average latency is around 0.3ms when there is no cache employed. When the SATA-flash cache is used the results show a small overhead of less than 0.023ms when the cache is cold, and for the PCI-e flash cache this overhead is less than 9μs. This overhead is mainly from looking up the requested block and finding the replacement block. Both operations are fast because dm-cache employs a radix tree for cache lookup which has a time complexity of $\mathrm{O}\big(\log n\big)$ where n is the maximum number of blocks in the the cache, and it maintains a linked-list-based LRU list for replacement. Once the requested block is fetched from the server, it is immediately returned to the upper layer in the IO stack while being stored into cache. When the cache is warm, the average latency drops to 0.107ms for the SATA flash cache and 0.23ms for the PCI-e flash cache, both of which match the raw flash read latencies (¡0.01ms slowdown) and are substantially faster than reading from the remote HDD.

For a sequential write workload (Figure 2.10), the average latency is around 0.4ms when flash cache is not used. When write-invalidate caching is used, the latencies are the same as when there is no cache since all the writes still need to be serviced by the remote server. When write-through or write-back caching is used, the latencies drop drastically, 0.06ms for the SATA flash and 0.05ms for the PCIe flash, because writes can be returned once they are stored by the flash cache. Their performance matches the raw device latencies with negligible difference (¡0.008ms slowdown) because the overhead introduced by cache lookup and finding the replacement block is small. For a random write workload (Figure 2.10), the latency difference between write-through/write-back and no-cache/write-invalidate is even more drastic, because the HDD-based back-end performs much worse for random writes than sequential writes while the flash's performance remains almost the same.

In summary, the above results confirm that the overhead introduced by dm-cache is small and insignificant even compared to the raw latencies of flash devices, thereby further verifying the feasibility of flash-based caching with software-based cache management.

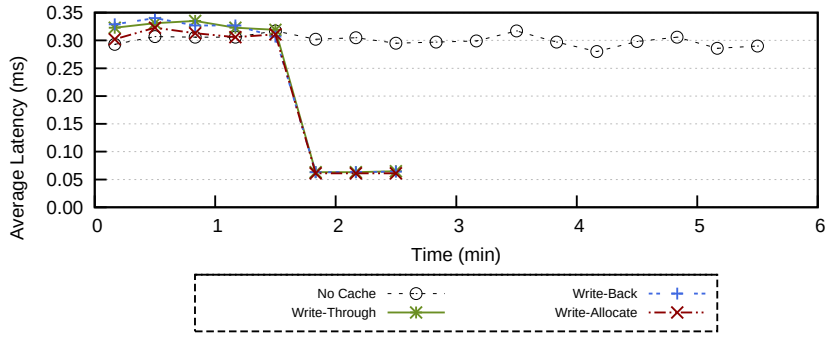
2.6 Latency Analysis

With the understanding of how different cache policies affect hit rate, we further study their impact on actual IO performance by analysing the IO latencies from caching different type of workloads. We use benchmarks to produce specific patterns of workloads and real trace to analyze IO performance of real-world workloads.

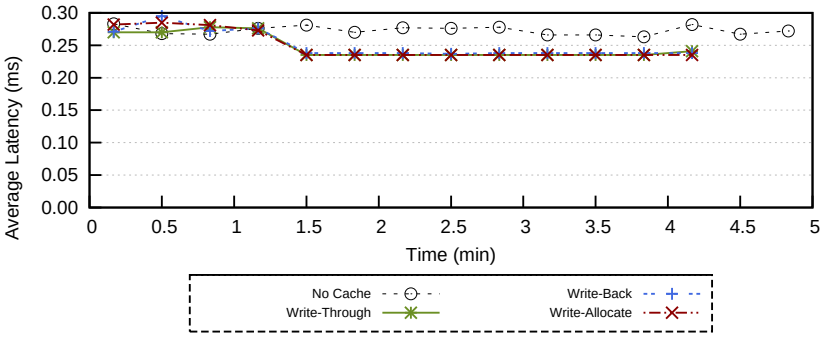
2.6.1 FIO Benchmark

Flexible IO (*fio*) is a versatile IO workload generator [fio]. It is used to create read and write intensive workloads of sequential and random patterns. Workloads are issued the raw storage device using direct IOs so that the impacts of file system and memory cache are avoided in this performance analysis. Each workload exercises 1GB of data and is repeated four times. We study the IO latencies for each workload using three different cache policies (write-through, write-allocate, and write-back)compared to the scenario where flash cache is not employed.

For a sequential read workload (Figure 2.11), the average latency is around 0.3 ms when there is no cache employed. When the SATA-flash cache is used, the results show a small overhead of less than 0.05 ms during the warm up time. Once the cache is warm, the average latency drops to 0.05 ms for the SATA flash cache and 0.23 ms for the PCIe flash cache, which match the raw flash read latency. For a sequential



(a) SATA SSD cache

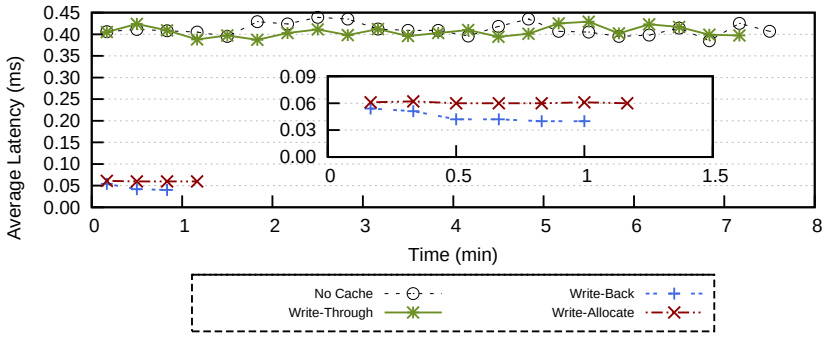


(b) PCIe SSD cache

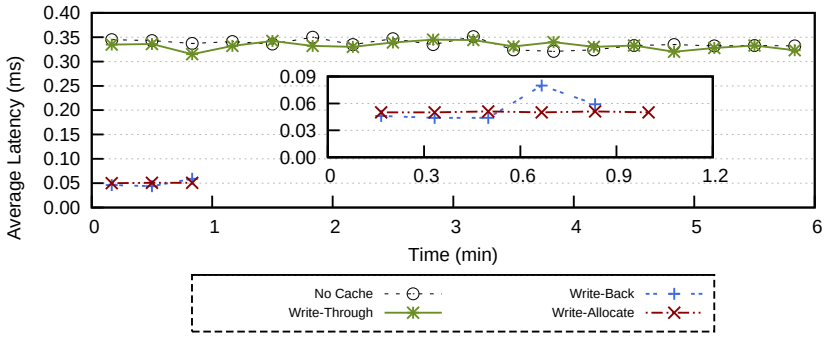
Figure 2.11: Sequential Reads

write workload (Figure 2.12), the average latency is around 0.4 ms when flash cache is not used. When write-through caching is used, the latencies are the same as when there is no cache since all the writes still need to be serviced by the remote server. When write-back or write-allocate caching is used, the latencies drop drastically, 0.06 ms for the SATA flash and 0.05 ms for the PCIe flash, because writes can be returned once they are serviced by the flash cache.

For a sequential read/write workload (Figure 2.13), the average latency is similar as sequential write when flash cache is not used, because the writes are scheduled together with reads causing less prefetching/batching for reads. When write-through caching is used latencies drops a little after cache is warm since all reads are ser-



(a) SATA SSD cache

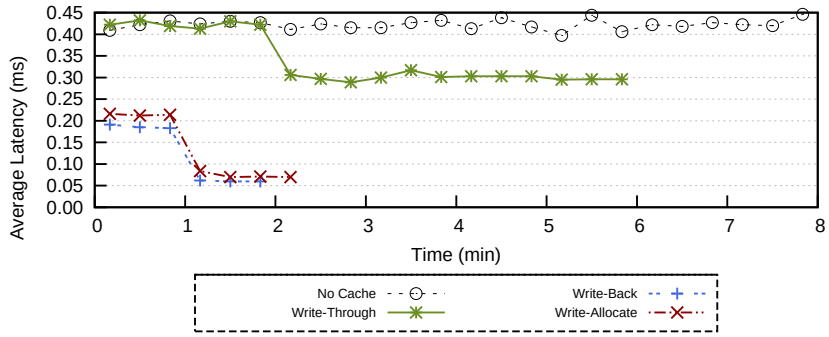


(b) PCI SSD cache

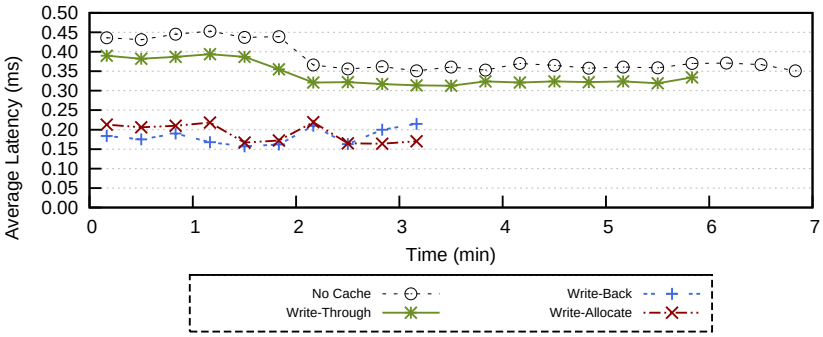
Figure 2.12: Sequential Writes

viced from cache. When write-back or write-allocate caching is used latencies drop substantially to 0.2 ms during warm up for SATA and PCIe flash, once the cache is warm we can see average latencies of 0.09 ms for SATA flash and 0.17 ms for PCIe, which reflect the raw flash latencies on both flash devices.

For a random read workload (Figure 2.14(a)), the average latency is similar as sequential reads, because the IOs are re-schedule before being dispatched to iSCSI server resulting sequential workload at the server side. For a random write workload (Figure 2.15), the latency difference between write-back/write-allocate and no-cache/write-through is even more drastic, because the HDD-based backend performs much worse for random writes than sequential writes.



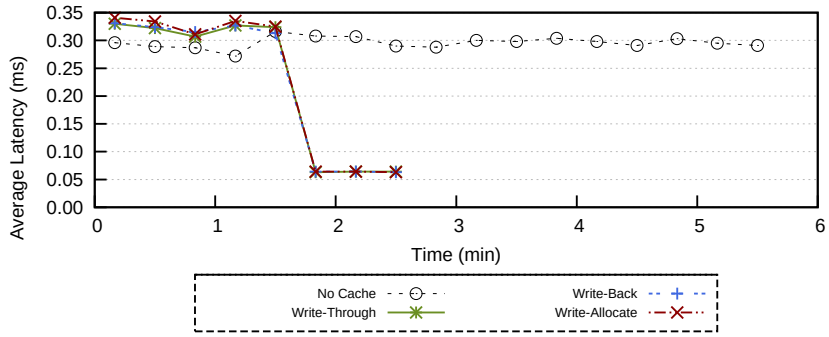
(a) SATA SSD cache



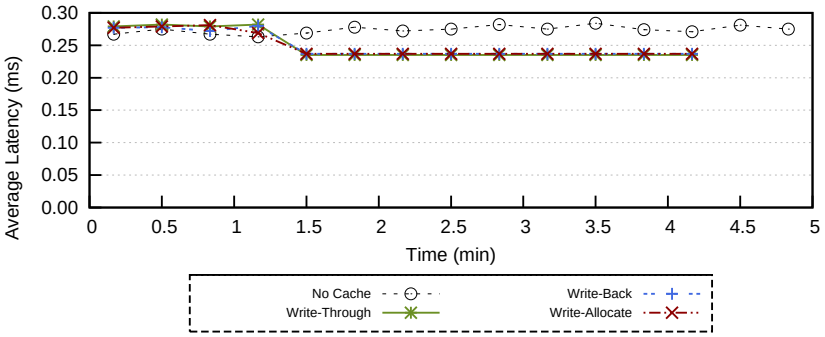
(b) PCIe SSD cache

Figure 2.13: Sequential Reads-Writes

For a random read/write workload (Figure 2.16), the average latency is around 1.2 ms when flash cached is not used. When write-through caching is used, the latencies are the same as when there is no cache since all IOs still need to be serviced by remote server. When write-back caching is used, the latencies drops drastically even at warm up since remote server only serve reads while writes are serviced from cache, once the cache is warm all IOS are serviced from cache generating 0.05 ms average latency. Finally when write-allocate caching is used, the latencies during warm up remain the same as when there is no cache involved, since read & writes are being forwarded to remote server, after warm-up average latencies are 0.05 ms.



(a) SATA SSD cache

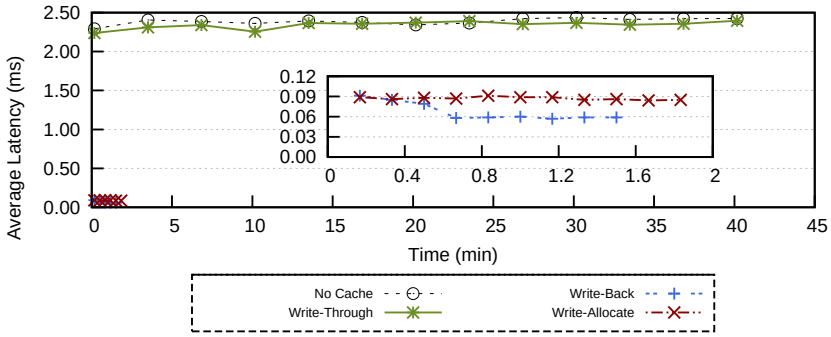


(b) PCIe SSD cache

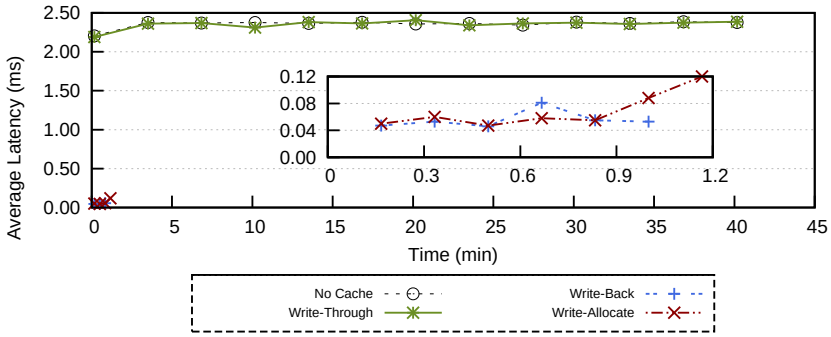
Figure 2.14: Random Reads

2.7 Write Policy Analysis

As shown in Section 2.3.3, a cloud workload can have a substantial amount of writes. This observation is also confirmed by related work [LPGM08, NDR08], which can be attributed to the fact that modern computer systems are getting larger memories which can cache reads in memory but do not buffer writes for too long due to durability concerns. Therefore, the choice of a write caching policy is important and it has implications on both performance and data durability. This section studies the impact of different write cache policies, where we use dm-cache-sim to study the impact on cache hit rate using long-term traces and use dm-cache to evaluate the impact on IO performance using real experiments driven by shorter traces.



(a) SATA SSD cache

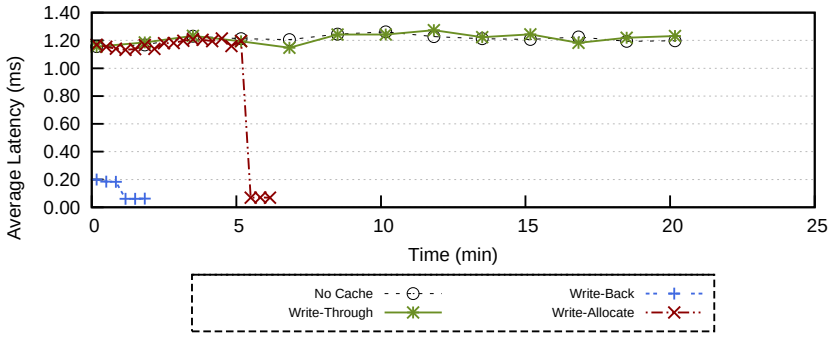


(b) PCIe SSD cache

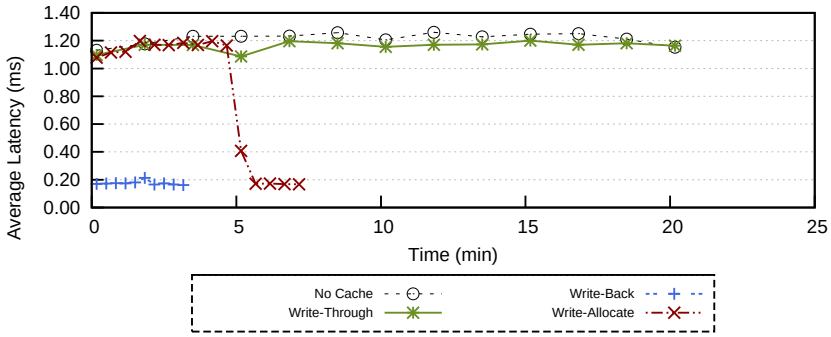
Figure 2.15: Random Writes

2.7.1 IO Latency

The various write caching policies impact IO latencies differently. If there is enough locality in writes, a policy that retains writes in cache (i.e., write-through or write-back) can speed up the IOs including both reads and writes that hit the cached blocks, compared to another policy that does not retain writes (i.e., write-invalidate). Otherwise, the limited cache capacity can be wasted which slows down the IOs that experience conflict misses. Comparing write-through policy to write-back policy, they exhibit the same behavior in terms of the cache hit rates but *not necessarily* the IO performance. Although writes can be returned as soon as they are stored in cache in both policies, the IOs that have to be serviced by the server experience



(a) SATA SSD cach



(b) PCIe SSD cach

Figure 2.16: Random Reads-Writes

different latencies. With the write-through policy, all the writes have to be sent to the server right away, while with the write-back policy, writes can be delayed and the following writes that hit the cached dirty data can be absorbed completely by the cache. Therefore, the server experiences a higher load under the write-through policy which in turn affects the performance of the clients. This difference can be significant in a highly consolidated environment such as a cloud system.

In order to evaluate the performance impact of different write caching policies, we consider two real workloads taken from the Web server and the Moodle server traces described in Section 2.3.3, which are relatively more write-intensive and read-intensive respectively. One typical day of workload was extracted from each trace

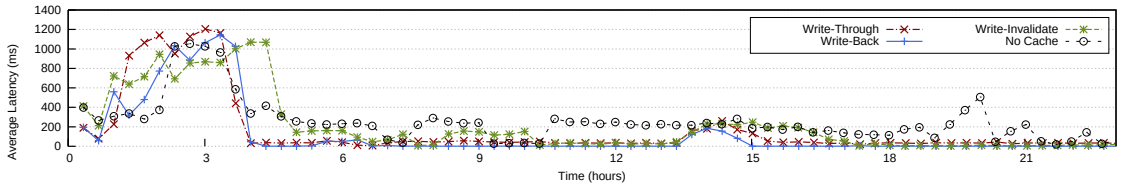
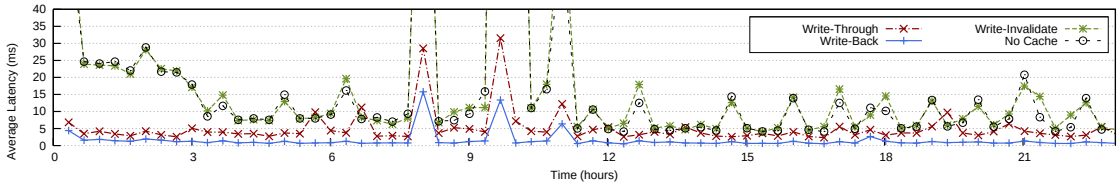
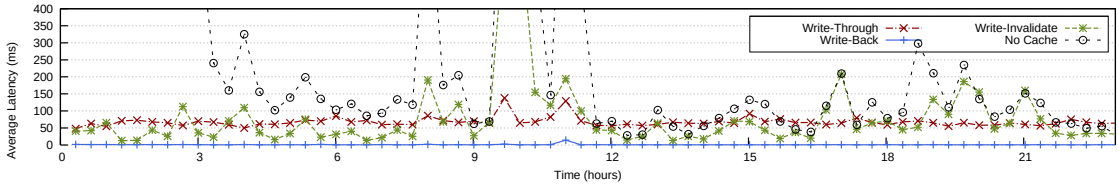


Figure 2.17: Performance of a read-intensive workload using different write caching policies



(a) Single Client



(b) Three Clients

Figure 2.18: Performance of a write-Intensive workload using different write caching policies

and replayed using *btreplay* at a 20-fold speedup in the environment specified in Section 2.3.4. While the accelerated replay makes the replayed workload more intensive than the original one, it is still a reasonable setup because, *1)* on a typical cloud VM host there can be well above 20 VMs running concurrently; *2)* the original trace would have also been more intensive on its own if there was a flash caching deployed to speed up its IOs.

Read-intensive Trace

First, we replayed a one-day read-intensive workload extracted from the Moodle server trace, which has a 20GB total working set size and consists of 65% reads and 35% writes. Figure 2.17 shows the average IO latencies measured every 20 minutes during the experiment using dm-cache with different write policies. The latencies from native iSCSI without dm-cache are also provided as a reference.

Initially, it takes around 5 hours to warm up the cache, during which the different write policies offer similar performance in term of latency because the performance is dominated by reads that miss the cache and have to be serviced by the server. Note that the *No Cache* case also exhibits a warm-up phase, although it does not employ a client-side flash cache, because the memory caches involved in this distributed storage system also need to be warmed up initially.

During the rest of the experiment, as the flash cache is warmed up to serve the reads, the difference in the write policy shows up where the write-back policy consistently outperforms the other policies. The IO latencies from both write-back and write-through policies are lower than write-invalidate by 58ms and 23ms in average respectively, because writes can be returned immediately after they are stored in cache. However, because the write-through policy still submits all writes to server, it slows down the read misses that have to be serviced by the server, although the latencies of writes are hidden to the client. Hence, the IO latencies from the write-through policy are higher than the write-back policy by 35ms (247%) in average.

Write-intensive Trace

The second experiment considers a one-day write-intensive workload extracted from the Web server trace, which has a total of 10GB WSS and consists of 15% read and

85%writes. We expect to see a larger performance difference among the different write caching policies compared to the above read-intensive trace.

Figure 2.18(a) shows the IO latencies measured every 20 minutes during the trace replay. For the *No Cache* and *Write Invalidate* cases, it also takes around 5 hours to warm up the caches. In contrast, the *Write Through* and *Write Back* cases do not exhibit a warm-up phase, because most of the IOs in this write-intensive workload can be directly serviced from the flash cache. The *Write Back* and *Write Through* policies present latencies lower that the case of *Write Invalidate* by 19ms and 3ms respectively. More importantly, throughout the experiment, the *Write Back* policy's IO latencies are lower than the *Write Through* policy by 3.5ms (230%) in average, mostly because it effectively reduces the server IO load and allows the IOs that have be serviced by the server to complete faster.

In order to evaluate the different write caching policies in a highly consolidated cloud environment, we employ three storage clients that share the same storage server in the next experiment, where each client replays a different day of the write-intensive Web server trace. In this more typical scenario, we can appreciate the substantial improvement made by the write-back caching: it's IO latencies are lower than the write-through caching by 67ms (5991%) in average. In fact, the performance of write-through caching is slowed down to the same level of the much simpler write-invalidate caching, with only 17ms improvement.

2.7.2 Server Load

As shown in the above experimental results, the write caching policies do exhibit evident differences in their impacts to a workload's IO performance. In particular, the difference between write-through and write-back can be significant. Although

both can hide the latency for writes, the difference in server IO load does impact the client-side performance substantially. As a further validation of these observations, we extend this write policy analysis to the entire traces using the dm-cache-sim simulator. However, instead of collecting hit rates, which are always the same between write-through and write-back, we collect the number of IO requests that are serviced by the server, which is the server IO load during these long-term traces.

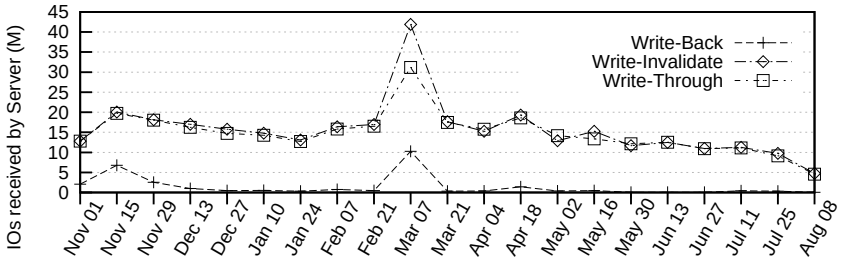


Figure 2.19: Server IO load for Web server trace

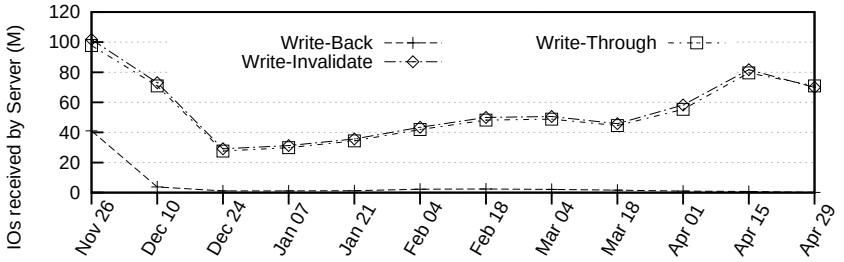


Figure 2.20: Server IO load for Moodle server trace

Figures 2.19-2.22 illustrate how the server load varies over the entire duration of the four FIU traces. All of them show that the write-back policy always results in substantially lower server load than the write-through policy. The largest improvement is from the *Bear file server* trace (Figure 2.22), which shows a 94% reduction on IO load. The smallest improvement is from the *Buffalo file server* trace (Figure 2.21), which shows a 52% reduction on IO load.

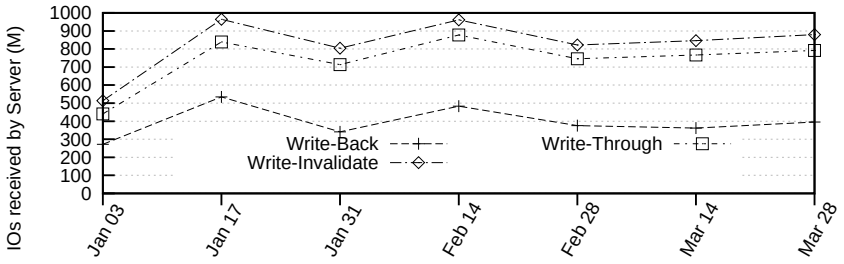


Figure 2.21: Server IO load for Buffalo server trace

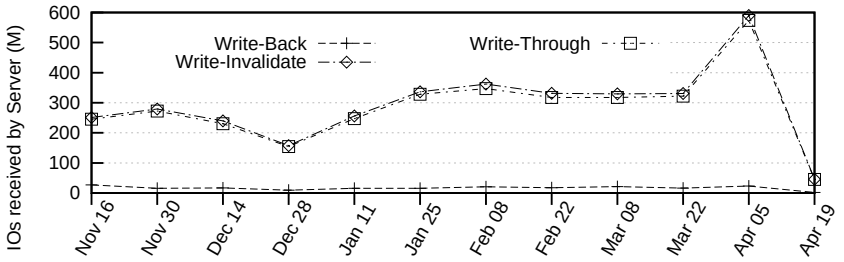


Figure 2.22: Server IO load for Bear server trace

Figure 2.23 shows the IO load on the storage server for the *Cloud VPS* traces, where each trace is replayed separately. In general we can see that the write-back policy still achieves the lowest IO load on the server, and the reduction varies from 21% to 83% compared to the write-through policy.

2.8 Persistency Analysis

2.8.1 Persistency Overhead

With the understanding of the impact on hit rate and IO latency of the different cache policies, we want to further analyze the overhead associated with making the cache persistent—although cached data blocks are always persistently stored on flash, the metadata of these blocks, including the source-to-cache address mappings

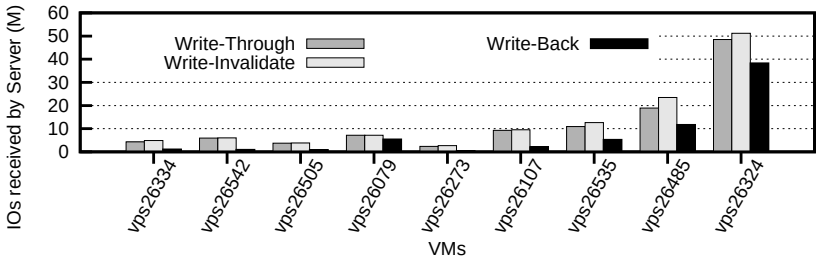


Figure 2.23: Server IO load for CloudVPS traces

and valid and dirty bits, also need to be considered in terms of their persistency. Storing the metadata persistently on flash allows the cached data to be reused after the storage client reboots, but it incurs more overhead. Moreover, if the write-back policy is used, the metadata of dirty blocks must be stored persistently; otherwise, these locally modified data will be lost after a reboot. Note that even if the storage client is completely lost, a persistent flash cache can be physically moved to a different client to reuse or recover the cached data. Based on the above considerations, we study two different persistency configurations for a flash cache.

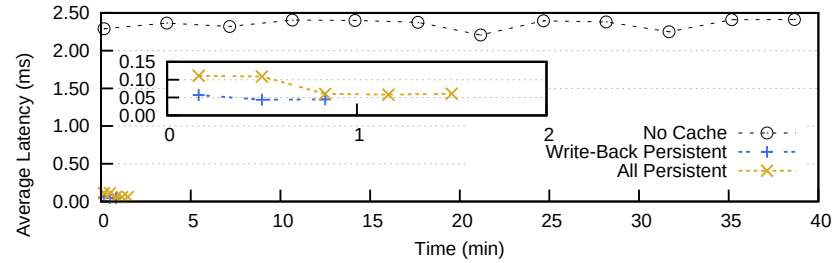
- *All-persistent*: The metadata of all cached blocks are persistently stored on the flash.
- *Write-back-persistent*: The metadata of only the dirty cache blocks are persistently stored on the flash.

To make a flash cache persistent, metadata updates need to be committed to the cache upon cache insertions, replacements, and invalidations. Our current implementation for making *dm-cache* persistent is quite straightforward. A metadata update is written to the flash at the same time of the corresponding cache insertion or replacement (but cache invalidations require only metadata updates and no data updates). The data and metadata updates are issued in parallel and the original

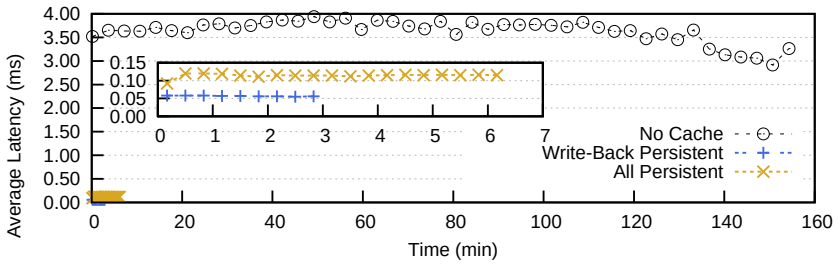
IO request received by dm-cache is returned only when both are committed to the cache. The IO latency is hence determined by the slower one between the data and metadata updates. Although flash devices typically have good internal parallelism to handle concurrent IOs, additional writes introduced by the metadata updates may degrade the performance of flash caching because writes tend to be slower than reads and get amplified due to the need of garbage collection.

More efficient handling of metadata update is possible but not trivial. For example, it is possible to combine the data and metadata updates in a single write, but the metadata is typically small and requires the update on a partial page. Related work [SSZ12] proposed to store the metadata in the out-of-band (OOB) area of a flash page on the device, but it requires changing the device's FTL and occupies the limited OOB area which is commonly used for important error correction. In addition to the potential slowdown, storing metadata in flash cache also reduces the size available for data caching; in our experiments, using a 120GB flash device total size, 1GB of the flash capacity needs to be reserved for metadata storage.

Figure 2.8.1 shows the IO latencies for various persistency configurations when handling a random read/write (50% reads and 50% writes) workload of different sizes generated by the *fio* benchmark. The write-back policy is used for both persistency configurations. When the workload is small (4GB random reads/writes with 1GB of WSS), the overhead of persistency is small (around 0.03ms); but when the workload is larger (10GB random reads/writes with 5GB of WSS), the overhead grows to 0.06ms (101.8%) as the addition metadata updates slow down the other cache accesses.



(a) 1GB random reads/writes



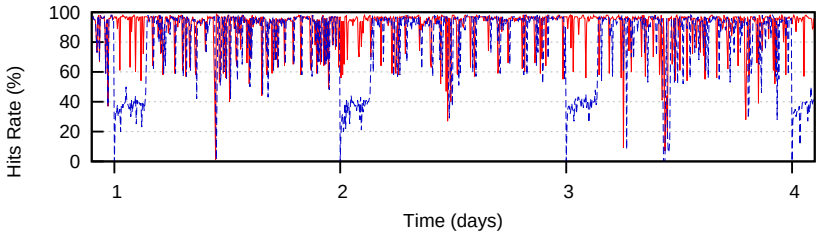
(b) 10GB random reads/writes

Figure 2.24: Persistency overhead with fio

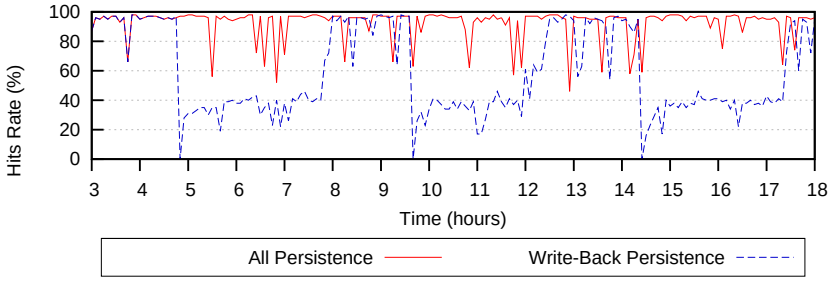
2.8.2 Persistency Benefits

Having a persistent cache allows the client to continue with a warm cache after it reboots or recover from a crash. In contrast, with a non-persistent cache, the client has to flush all the cached data after it comes back and warms up the cache from scratch, which may lead to substantial compulsory misses. We study this performance improvement from a persistent cache by analyzing the cache hit rate from the two different configurations, *all-persistent* and *write-back-persistent* using dm-cache-sim, while considering different reboot/crash frequencies (daily and hourly).

Figure 2.25(a) shows the results from a workload extracted from the Web server trace, assuming the client reboots or recovers upon the start of every day in the experiment. The results show that upon every reboot/recovery, the write-back-persistent configuration has to warm up the cache again, which in average takes 5 hours, whereas the all-persistent configuration always enjoys a warm cache despite



(a) Daily crashes



(b) Hourly crashes

Figure 2.25: Cache hit rate changes over time with different persistency configurations

of the reboots or crashes. In average, the hit rate of all-persistent configuration is higher than the write-back-persistent configuration by 7.97% in this experiment. Note that the hit rate drops in the middle of day which happens to both configurations and is caused by the change of data locality.

Figure 2.25(b) shows the results from a workload extracted from the Moodle server trace, assuming the client reboots or recovers upon the start of every 5th hour in the experiment. For this workload, using the write-back-persistent configuration, it takes in average 3 hours to warm up the cache again after a reboot/crash, and as a result the hit rate is lower by 27.66% than the all-persistent configuration which has a warm cache persisting across reboots/crashes.

The above cost and benefit analysis shows a clear tradeoff. Making a flash cache entirely persistent slows down IO latencies during normal operations but improves hit rates after client reboots. This decision should be made based on the expected

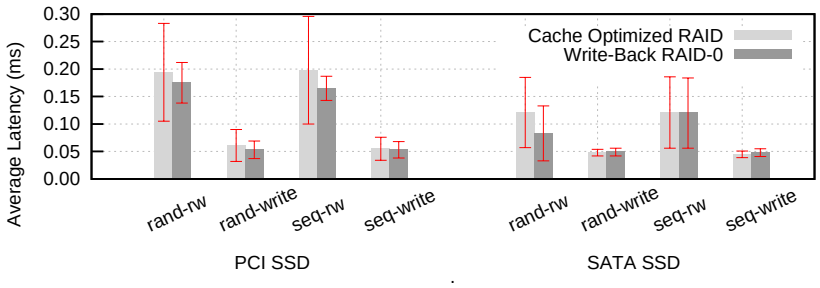


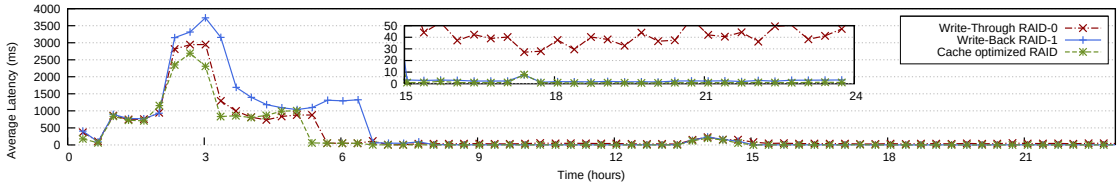
Figure 2.26: Overhead of cache-optimized RAID

client failure rate for a given cloud system.

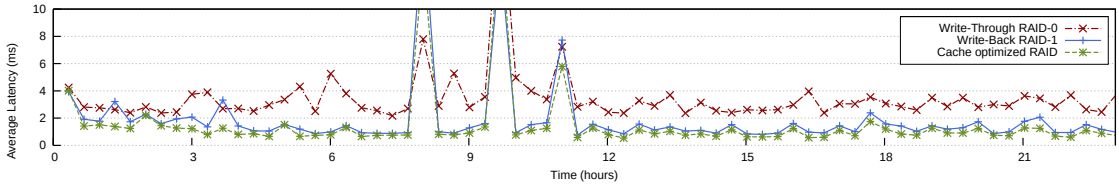
2.9 Reliability

While the persistent flash cache discussed in the previous section allows the cache to tolerate client restarts and crashes, it does not protect data against flash device failures, including memory cell failures that cannot be masked by the device controller and catastrophic whole-device or whole-chip failures. This concern for data reliability is the reason why flash caching is commonly used in write-through mode, by submitting writes to the storage server while caching them on the flash device, instead of the write-back mode in which writes are delayed in cache without immediately submitted to server. However, as shown in Section 2.7, write-back caching can substantially improve the storage client's performance and reduce the storage server's load. This conflicts presents a challenge to the effective use of flash caching.

RAID is a classic technique used to tolerate catastrophic failures for hard-disk storage, and has been recently studied for flash storage [JMBR11, BKPM10]. However, compared to the level of RAID extensively employed on the storage server, the use of RAID for flash caching faces two major limitations. *First*, the cost of using RAID for a flash cache is substantially more expensive than the cost on the storage



(a) Read-intensive workload



(b) Write-intensive workload

Figure 2.27: Performance of different reliability configurations

server. Following the general principle of forming an effective storage hierarchy, for a storage layer to be fast enough as a cache for the underlying layer, it has to use a technology that is typically much more expensive in terms of per unit size cost. *Second*, using RAID to improve the reliability for a flash cache is at conflict with the other objectives, particularly performance—more redundancy leads to less capacity for storing data localities, and endurance—more redundancy also leads to more wear-out to the flash of the same size.

To address the above limitations, we propose a new *cache-optimized RAID* technique by exploiting different levels of reliability needs for clean data and dirty data to improve cache utilization and reduce its cost. On one hand, clean data in the cache do not require extra redundancy, and their flash pages can employ RAID-0 across the participating flash devices to provide only performance improvement via striping. On the other hand, dirty data in the cache must be provided the same level of reliability as the primary storage, so they will employ higher levels of RAID to tolerate different types of failures. In this way, the cost of using RAID to provide fault tolerance can be minimized by introducing only the necessary redundancy into

a flash cache, and this approach has the potential to make a write-back cache reliable and affordable. For the same reason, the adversary impact of using RAID to cache performance and wear-out can also be minimized. Furthermore, the tradeoff between these conflicting objectives can be flexibly adjusted by tuning the amount of dirty data kept in cache.

We have implemented this cache-optimized RAID technique in dm-cache. In this implementation, reads are striped among the flash devices in a RAID-0 fashion for performance improvement while writes are replicated among the devices in a RAID-1 fashion for reliability improvement. For replacement, a read replaces the LRU block considering all devices in the RAID group, a write is replicated across the devices using the LRU block on each device.

The rest of this section evaluates our proposed cache-optimized RAID technique. First we study the overhead by comparing the *cache-optimized RAID* with a vanilla write-back cache layered on top native Linux RAID-0 (*Write-back RAID-0*). Because the *Write-back RAID-0* does not provide any data redundancy, this experiment evaluates the overhead incurred by replicating the dirty cached blocks in our *cache-optimized RAID*. We employed two identical flash devices on the client for the RAID configurations and used *fio* to generate different workload patterns for the experiment. The results in Figure 2.26 show that this overhead is small (less than 9.1μs (9%) increase in IO latency).

We further analyze the performance of the cache-optimized RAID for real-world workloads and compare it to the alternative options that also provide data reliability, including write-through caching on top of native Linux RAID-0 (*Write-through RAID-0*) and write-back caching on native RAID-1 (*Write-back RAID-1*). We consider the same two workloads used in Section 2.7.1, one read-intensive from the Moodle server trace and the other write-intensive from the Web server trace. Fig-

ure 2.27(a) shows the IO latencies for the read-intensive workload. In average, the *cache-optimized RAID* configuration's latencies are lower than the *Write-through RAID-0* and *Write-back RAID-1* configurations by 63ms (26%) and 172ms (72%) respectively. Because of the slower performance of the *write-back RAID-1*, from replicating every block and half-reduced capacity, its warm-up time is also stretched longer than the other two configurations. Figure 2.27(b) shows the latencies for the write-intensive workload, where the *cache-optimized RAID* configuration's latencies are again lower than *Write-through RAID-0* and *Write-back RAID-1* by 1.97ms (135%) and 0.23ms (23%) in average, respectively.

2.10 Summary

This chapter provides answers to several key questions that determine how to effectively use flash caching in cloud systems. *How to size the flash caches?* A good cache size should be able to hold the WSS of a given workload, and we analyzed a set of representative traces to understand the typical WSSes of cloud workloads. *How to choose the write caching polices?* There is always trade-off among different policies in term of performance, reliability, and consistency. Different from the conclusion from related work, our results show that write-back caching can substantially outperform write-through caching due to the reduction of server IO load. *Is it necessary to make a flash cache persistent across client restarts and crashes?* Making a flash cache persistent across client restarts incurs small overhead, but it can save the cache warm-up phase which typically lasts hours according to our analysis. *How to improve the reliability of a flash cache so as to tolerate device-level failures?* To address the reliability issue of write-back caching, we propose a new cache-optimized data redundancy (RAID) technique which minimizes the RAID overhead by intro-

ducing redundancy to only cache dirty data and shows to be significantly faster than traditional RAID and write-through caching.

We have collected large amounts of traces from both production public and private clouds. To effective analyze these traces and understand the impacts of various key caching policies and cache allocation approaches, we have developed a user-space cache simulator. Finally, we have also implemented these techniques in a real cloud caching framework, dm-cache. The results confirm that cloud computing systems are a good target for flash caching, but the locality in writes must be effectively utilized. The results also show that the working-set sizes of real cloud workloads can be accurately predicted online and used to guide efficient dynamical cache allocation.

CHAPTER 3

On-demand Space Allocation

3.1 Introduction

Host-side flash caching employs flash-memory-based storage on a virtual machine (VM) host as the cache for its remote storage to exploit the data access locality and improve the VM performance. It has received much attention in recent years [BLM+12, ioC, HAWS13, AZ14], which can be attributed to two important reasons. First, as the level of consolidation continues to grow in cloud computing systems, the scalability of shared VM storage servers becomes a serious issue. Second, the emergence of flash-memory-based storage has made flash caching a promising option to address this IO scalability issue, because accessing a local flash cache is substantially faster than accessing the remote storage across the network.

However, due to the capacity and cost constraints of flash devices, the amount of flash cache that can be employed on a host is much limited compared to the dataset sizes of the VMs, particularly considering the increasing data intensity of the workloads and increasing number of workloads consolidated to the host via virtualization. Therefore, to fulfill the potential of flash caching, it is important to allocate the shared cache capacity among the competing VMs according to their actual demands. Moreover, flash devices wear out by writes and face serious endurance issues, which are in fact aggravated by the use for caching because both the writes inherent in the workload and the reads that miss the cache induce wear-out [YPGT13, HWC+13]. Therefore, the cache management also needs to be careful

not to admit data that are not useful to workload performance and only damage the endurance.

We propose CloudCache to address the above issues in flash caching through *on-demand cache management*. Specifically, it answers this challenging question, *how to allocate a flash cache to VMs according to their cache demands?* Flash cache workloads depend heavily on the dynamics in the upper layers of the IO stack and are often unfeasible to profile offline. The classic working set model studied for processor and memory cache management can be applied online, but it does not consider the reuse behavior of accesses and may admit data that are detrimental to performance and endurance. To address this challenge, we propose a new cache demand model, *Reuse Working Set (RWS)*, to capture the data that have good temporal locality and are essential to the workload's cache hit ratio, and use the RWS size (*RWSS*), to represent the workload's cache demand. Based on this model, we further use prediction methods to estimate a workload's cache demand online and use new cache admission policies to admit only the RWS into cache, thereby delivering a good performance to the workload while minimizing the wear-out. CloudCache is then able to allocate the shared cache capacity to the VMs according to their actual cache demands.

We provide a practical implementation of CloudCache based on block-level virtualization [HZ06]. It can be seamlessly deployed onto existing cloud systems as a drop-in solution and transparently provide caching and on-demand cache management. We evaluate it using a set of long-term traces collected from real-world cloud systems [AZ14]. The results show that RWSS-based cache allocation can substantially reduce cache usage and wear-out at the cost of only small performance loss in the worst case. Compared to the WSS-based cache allocation, the RWSS-based method reduces a workload's cache usage by up to 76%, lowers the amount

of writes sent to cache device by up to 37%, while delivering the same IO latency performance. Compared to the case where the VM can use the entire cache, the RWSS-based method saves even more cache usage while delivering an IO latency that is only 1% slower at most.

To the best of our knowledge, CloudCache is the first to propose the RWSS model for capturing a workload's cache demand from the data with good locality and for guiding the flash cache allocation to achieve both good performance and endurance. While the discussion in the chapter focuses on flash-memory-based caches, we believe that the general CloudCache approach is also applicable to new nonvolatile memory (NVM) technologies (e.g., PCM, 3D Xpoint) which will likely be used as a cache layer, instead of replacing DRAM, in the storage hierarchy and will still need on-demand cache allocation to address its limited capacity (similarly to or less than flash) and endurance (maybe less severe than flash).

The rest of the chapter is organized as follows: Section 3.2 and Section 3.3 present the motivations and architecture of CloudCache, and Section 3.4 describe the on-demand cache allocation including evaluation results.

3.2 Motivations

The emergence of flash-memory-based storage has greatly catalyzed the adoption of a new flash-based caching layer between DRAM-based main memory and HDD-based primary storage [BLM+12, ioC, AZ14, MZM+14]. It has the potential to solve the severe scalability issue that highly consolidated systems such as public and private cloud computing systems are facing. These systems often use shared network storage [KHSB02, nbd] to store VM images for the distributed VM hosts, in order to improve resource utilization and facilitate VM management (including

live VM migration [CFH+05, NLH05]). The availability of a flash cache on a VM host can accelerate the VM data accesses using data cached on the local flash device, which are much faster than accessing the hard-disk-based storage across network. Even with the increasing adoption of flash devices as primary storage, the diversity of flash technologies allows the use of a faster and smaller flash device (e.g., single-level cell flash) as the cache for a slower but larger flash device (e.g., multi-level cell flash) used as primary storage.

To fulfill the potential of flash caching, it is crucial to employ *on-demand cache management*, i.e., allocating shared cache capacity among competing workloads based on their demands. The capacity of a commodity flash device is typically much smaller than the dataset size of the VMs on a single host. How the VMs share the limited cache capacity is critical to not only their performance but also the flash device endurance. On one hand, if a workload's necessary data cannot be effectively cached, it experiences orders of magnitude higher latency to fetch the missed data from the storage server and at the same time slows down the server from servicing the other workloads. On the other hand, if a workload occupies the cache with unnecessary data, it wastes the valuable cache capacity and compromises other workloads that need the space. Unlike in CPU allocation where a workload cannot use more than it needs, an active cache workload can occupy all the allocated space beyond its actual demand, thereby hurting both the performance of other workloads and the endurance of flash device.

S-CAVE [LML+13] and vCacheShare [MZM+14] studied how to optimize flash cache allocation according to a certain criteria (e.g., a utility function), but they cannot estimate the workloads' actual cache demands and thus cannot meet such demands for meeting their desired performance. HEC [YPGT13] and LARC [HWC+13] studied cache admission policies to reduce the wear-out damage caused by data

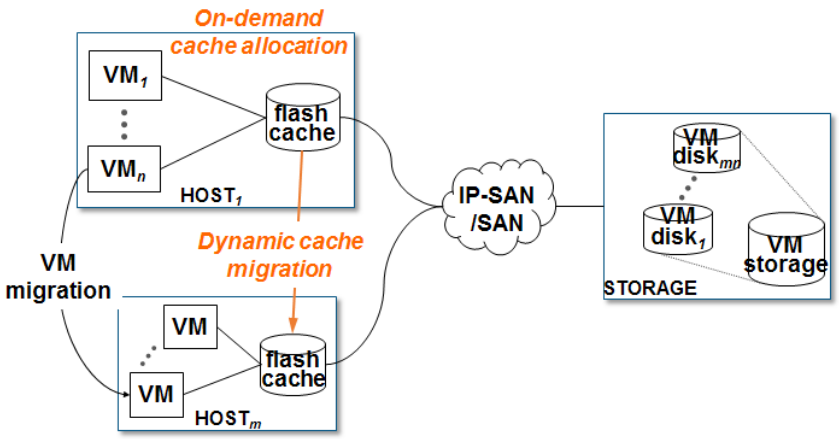


Figure 3.1: Architecture of CloudCache

with weak temporal locality, but they did not address the cache allocation problem. Bhagwat *et al.* studied how to allow a migrated VM to access the cache on its previous host [BPO+15], but they did not consider the performance impact to the VMs. There are related works studying other orthogonal aspects of flash caching, including write policies [KMR+13], deduplication/compression [LSD+14], and other design issues [HAWS13, AZ14].

3.3 Architecture

CloudCache supports *on-demand cache management* based on a typical flash caching architecture illustrated in Figure 3.1. The VM hosts share a network storage for storing the VM disks, accessed through SAN or IP SAN [KHSB02, nbd]. Every host employs a flash cache, shared by the local VMs, and every VM's access to its remote disk goes through this cache. CloudCache provides on-demand allocation of a flash cache to its local VMs and dynamic VM and cache migration across hosts to meet the cache demands of the VMs. Although our discussions in this chapter

focus on block-level VM storage and caching, our approaches also work for network file system based VM storage, where CloudCache will manage the allocation and migration for caching a VM disk file in the same fashion as caching a VM's block device. A VM disk is rarely write-shared by multiple hosts, but if it does happen, CloudCache needs to employ a cache consistency protocol [NWO88], which is beyond the scope of this chapter.

CloudCache supports different write caching policies: (1) *Write-invalidate*: The write invalidates the cached block and is submitted to the storage server; (2) *Write-through*: The write updates both the cache and the storage server; (3) *Write-back*: The write is stored in the cache immediately but is submitted to the storage server later when it is evicted or when the total amount of dirty data in the cache exceeds a predefined threshold. The write-invalidate policy performs poorly for write-intensive workloads. The write-through policy's performance is close to write-back when the write is submitted to the storage server asynchronously and the server's load is light [HAWS13]; otherwise, it can be substantially worse than the write-back policy [AZ14]. Our proposed approaches work for all these policies, but our discussions focus on the write-back policy due to limited space for our presentation. The reliability and consistency of delayed writes in write-back caching are orthogonal issues to this chapter's focus, and CloudCache can leverage the existing solutions (e.g., [KMR+13]) to address them.

In the next few sections, we introduce the two components of CloudCache, *on-demand cache allocation* and *dynamic cache migration*. As we describe the designs, we will also present experimental results as supporting evidence. We consider a set of block-level IO traces [AZ14] collected from a departmental private cloud as representative workloads. The characteristics of the traces are summarized in Table 3.1. These traces allow us to study long-term cache behavior, in addition to the

Trace	Time (days)	Total IO (GB)	WSS (GB)	Write (%)
Webserver	281	2,247	110	51
Moodle	161	17,364	223	13
Fileserver	152	57,887	1037	22

Table 3.1: Trace statistics

commonly used traces [msr] which are only week-long.

3.4 On-demand Cache Allocation

CloudCache addresses two key questions about on-demand cache allocation. First, *how to model the cache demand of a workload?* A cloud workload includes IOs with different levels of temporal locality which affect the cache hit ratio differently. A good cache demand model should be able to capture the IOs that are truly important to the workload's performance in order to maximize the performance while minimizing cache utilization and flash wear-out. Second, *how to use the cache demand model to allocate cache and admit data into cache?* We need to predict the workload's cache demand accurately online in order to guide cache allocation, and admit only the useful data into cache so that the allocation does not get overflown. In this section, we present the CloudCache's solutions to these two questions.

3.4.1 RWS-based Cache Demand Model

Working Set (WS) is a classic model often used to estimate the cache demand of a workload. The working set $WS(t, T)$ at time t is defined as the set of distinct (address-wise) data blocks referenced by the workload during a time interval $[t - T, t]$ [Den68]. This definition uses the principle of locality to form an estimate of the

set of blocks that the workload will access next and should be kept in the cache. The *Working Set Size* (WSS) can be used to estimate the cache demand of the workload.

Although it is straightforward to use WSS to estimate a VM's flash cache demand, a serious limitation of this approach is that it does not differentiate the level of temporal locality of the data in the WS. Unfortunately, data with weak temporal locality, e.g., long bursts of sequential accesses, are abundant at the flash cache layer, as they can be found in many types of cloud workloads, e.g., when the guest system in a VM performs a weekly backup operation. Caching these data is of little benefit to the application's performance, since their next reuses are too far into the future. Allowing these data to be cached is in fact detrimental to the cache performance, as they evict data blocks that have better temporal locality and are more important to the workload performance. Moreover, they cause unnecessary wear-out to the flash device with little performance gain in return.

To address the limitation of the WS model, we propose a new cache-demand model, *Reuse Working Set*, $RWS_N(t, T)$, which is defined as the set of distinct (address-wise) data blocks that a workload has *reused* at least N times during a time interval $[t - T, t]$. Compared to the WS model, RWS captures only the data blocks with a temporal locality that will benefit the workload's cache hit ratio. When $N = 0$ RWS reduces to WS. We then propose to use *Reuse Working Set Size* (*RWSS*) as the estimate of the workload's cache demand. Because RWSS disregards low-locality data, it has the potential to more accurately capture the workload's actual cache demand, and reduce the cache pollution and unnecessary wear-out caused by such data references.

To confirm the effectiveness of the RWS model, we analyze the MSR Cambridge traces [msr] with different values of N and evaluate the impact on cache hit ratio, cache usage—the number of cached blocks vs. the number of IOs received by cache,

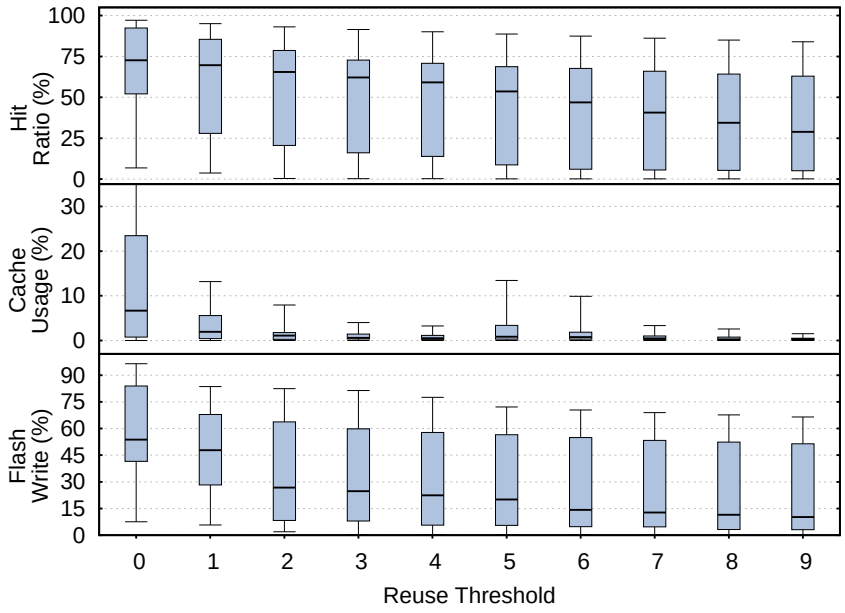


Figure 3.2: RWS analysis using different values of N

and flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. We assume that a data block is admitted into the cache only after it has been accessed N times, i.e., we cache only the workload's RWS_N. Figure 3.2 shows the distribution of these metrics from the 36 MSR traces using box plots with whiskers showing the quartiles. Increasing N from 0, when we cache the WS, to 1, when we cache the RWS_1, the median hit ratio is reduced by 8%, but the median cache usage is reduced by 82%, and the amount of flash writes is reduced by 19%. This trend continues as we further increase N.

These results confirm the effectiveness of using RWSS to estimate cache demand—it is able to substantially reduce a workload's cache usage and its induced wear-out at a small cost of hit ratio. A system administrator can balance performance against cache usage and endurance by choosing the appropriate N for the RWS model. In general, $N = 1$ or 2 gives the best tradeoff between these objectives. (Similar obser-

vations can be made for the traces listed in Table 3.1.) In the rest of this chapter, we use $N = 1$ for RWSS-based cache allocation. Moreover, when considering a cloud usage scenario where a shared cache cannot fit the working-sets of all the workloads, using the RWS model to allocate cache capacity can achieve better performance because it prevents the low-locality data from flushing the useful data out of the cache.

In order to measure the RWSS of a workload, we need to determine the appropriate time window to observe the workload. There are two relevant questions here. First, how to track the window? In the original definition of process WS [Den68], the window is set with respect to the process time, i.e., the number of accesses made by the process, instead of real time. However, it is difficult to use the number of accesses as the window to measure a VM's WS or RWSS at the flash cache layer, because the VM can go idle for a long period of time and never fill up its window, causing the previously allocated cache space to be underutilized. Therefore, we use real-time-based window to observe a workload's RWSS.

The second question is how to decide the size of the time window. If the window is set too small, the observed RWS cannot capture the workload's current locality, and the measured RWSS underestimates the workload's cache demand. If the window is set too large, it may include the past localities that are not part of the workload's current behavior, and the overestimated RWSS will waste cache space and cause unnecessary wear-out. Our solution to this problem is to profile the workload for a period of time, and simulate the cache hit ratio when we allocate space to the workload based on its RWSS measured using different sizes of windows. We then choose the window at the "knee point" of this hit ratio vs. window size model, i.e., the point where the hit ratio starts to flatten out. This profiling can be performed periodically, e.g., bi-weekly or monthly, to adjust the choice of window size *online*.

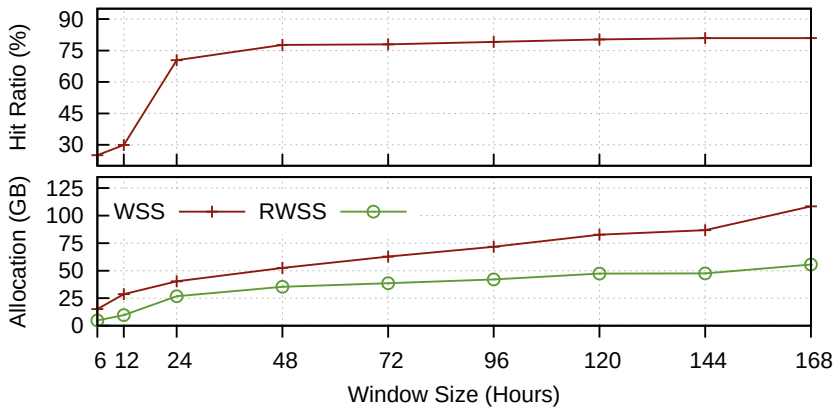


Figure 3.3: Time window analysis for the Moodle trace

We present an example of estimating the window size using two weeks of the Moodle trace. Figure 3.3 shows that the hit ratio increases rapidly as the window size increases initially. After the 24-hour window size, it starts to flatten out, while the observed RWSS continues to increase. Therefore, we choose between 24 to 48 hours as the window size for measuring the RWSS of this workload, because a larger window size will not get enough gain in hit ratio to justify the further increase in the workload's cache usage, if we allocate the cache based on the observed RWSS. In case of workloads for which the hit ratio keeps growing slowly with increasing window size but without showing an obvious knee point, the window size should be set to a small value because it will not affect the hit ratio much but can save cache space for other workloads with clear knee points.

3.4.2 Online Cache Demand Prediction

The success of RWSS-based cache allocation also depends on whether we can accurately predict the cache demand of the next time window based on the RWSS values observed from the previous windows. To address this problem, we consider the clas-

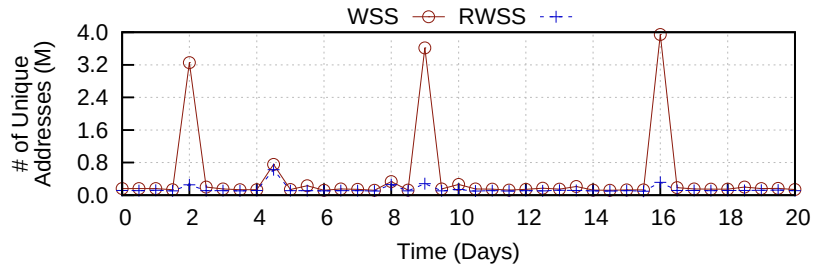
sic exponential smoothing and double exponential smoothing methods. The former requires a smoothing parameter α, and the latter requires an additional trending parameter β. The values of these parameters can have a significant impact on the prediction accuracy. We address this issue by using the self-tuning versions of these prediction models, which estimate these parameters based on the error between the predicted and observed RWSS values.

To further improve the robustness of the RWSS prediction, we devise filtering techniques which can dampen the impact of outliers in the observed RWSS values when predicting RWSS. If the currently observed RWSS is λ times greater than the average of the previous n observed values (including the current one), this value is replaced with the average. For example, n is set to 20 and λ is set to 5 in our experiments. In this way, an outlier's impact in the prediction is mitigated.

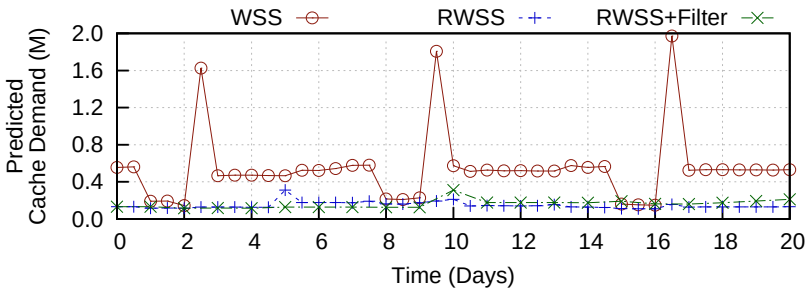
Figure 3.4 shows an example of the RWSS prediction for three weeks of the Webserver trace. The recurring peaks in the observed WSS in Figure 3.4(a) are produced by a weekly backup task performed by the VM, which cause the predicted WSS in Figure 3.4(b) to be substantially inflated. In comparison, the RWSS model automatically filters out these backup IOs and the predicted RWSS is only 26% of the WSS on average for the whole trace. The filtering technique further smooths out several outliers (e.g., between Day 4 and 5) which are caused by occasional bursts of IOs that do not reflect the general trend of the workload.

3.4.3 Cache Allocation and Admission

Based on the cache demands estimated using the RWSS model and prediction methods, the cache allocation to the concurrent VMs is adjusted accordingly at the start of every new time window—the smallest window used to estimate the RWSS of all



(a) Observed cache demand



(b) Predicted demand

Figure 3.4: RWSS-based cache demand prediction

the VMs. The allocation of cache capacity should not incur costly data copying or flushing. Hence, we consider *replacement-time enforcement* of cache allocation, which does not physically partition the cache across VMs. Instead, it enforces logical partitioning at replacement time: a VM that has not used up its allocated share takes its space back by replacing a block from VMs that have exceeded their shares. Moreover, if the cache is not full, the spare capacity can be allocated to the VMs proportionally to their predicted RWSSes or left idle to reduce wear-out.

The RWSS-based cache allocation approach also requires an *RWSS-based cache admission policy* that admits only reused data blocks into the cache; otherwise, the entire WS will be admitted into the cache space allocated based on RWSS and evict useful data. To enforce this cache admission policy, CloudCache uses a small portion

of the main memory as the staging area for referenced addresses, a common strategy for implementing cache admission [YPGT13, HWC+13]. A block is admitted into the cache only after it has been accessed N times, no matter whether they are reads or writes. The size of the staging area is bounded and when it gets full the staged addresses are evicted using LRU. We refer to this approach of staging only addresses in main memory as *address staging*.

CloudCache also considers a *data staging* strategy for cache admission, which stores both the addresses and data of candidate blocks in the staging area and manages them using LRU. Because main memory is not persistent, so more precisely, only the data returned by read requests are staged in memory, but for writes only their addresses are staged. This strategy can reduce the misses for read accesses by serving them from the staging area before they are admitted into the cache. The tradeoff is that because a data block is much larger than an address (8B address per 4KB data), for the same staging area, data staging can track much less references than address staging and may miss data with good temporal locality.

To address the limitations of address staging and data staging and combine their advantages, CloudCache considers a third *hybrid staging* strategy in which the staging area is divided to store addresses and data, and the address and data partitions are managed using LRU separately. This strategy has the potential to reduce the read misses for blocks with small reuse distances by using data staging and admitting the blocks with relative larger reuse distances by using address staging.

3.4.4 Evaluation

The rest of this section presents an evaluation of the RWSS-based on-demand cache allocation approach. CloudCache is created upon block-level virtualization by pro-

viding virtual block devices to VMs and transparently caching their data accesses to remote block devices accessed across the network (Figure 3.1). It includes a kernel module that implements the virtual block devices, monitors VM IOs, and enforces cache allocation and admission, and a user-space component that measures and predicts RWSS and determines the cache shares for the VMs. The kernel module stores the recently observed IOs in a small circular buffer for the user-space component to use, while the latter informs the former about the cache allocation decisions. The current implementation of CloudCache is based on Linux and it can be seamlessly deployed as a drop-in solution on Linux-based environments including VM systems that use Linux-based IO stack [BDF+03, kvm]. We have also created a user-level cache simulator of CloudCache to facilitate the cache hit ratio and flash write ratio analysis, but we use only the real implementation for measuring real-time performance.

The traces described in Section 3.3 are replayed on a real iSCSI-based storage system. One node from a compute cluster is set up as the storage server and the others as the clients. Each node has two six-core 2.4GHz Xeon CPUs and 24GB of RAM. Each client node is equipped with the CloudCache modules, as part of the Dom0 kernel, and flash devices (Intel 120GB MLC SATA-interface) to provide caching to the hosted Xen VMs. The server node runs the IET iSCSI server to export the logical volumes stored on a 1TB 7.2K RPM hard disk to the clients via a Gigabit Ethernet. The clients run Xen 4.1 to host VMs, and each VM is configured with 1 vCPU and 2GB RAM and runs kernel 2.6.32. The RWSS window size for the Webserver, Moodle, and Fileserver traces are 48, 24, and 12 hours, respectively. Each VM's cache share is managed using LRU internally, although other replacement policies are also possible.

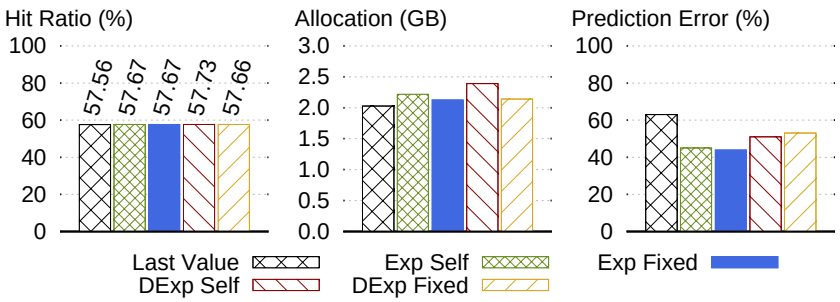


Figure 3.5: Prediction accuracy

Prediction Accuracy

In the first set of experiments we evaluate the different RWSS prediction methods considered in Section 3.4.2: (1) *Exp fixed*, exponential smoothing with $\alpha = 0.3$, (2) *Exp self*, a self-tuning version of exponential smoothing, (3) *DExp fixed*, double-exponential smoothing with $\alpha = 0.3$ and $\beta = 0.3$, (4) *DExp self*, a self-tuning version of double-exponential smoothing, and (5) *Last value*, a simple method that uses the last observed RWSS value as predicted value for the new window.

Figure 3.5 compares the different prediction methods using three metrics: (1) *hit ratio*, (2) *cache allocation*, and (3) *prediction error*—the absolute value of the difference between the predicted RWSS and observed RWSS divided by the observed RWSS. Prediction error affects both of the other two metrics—under-prediction increases cache misses and over-prediction uses more cache. The figure shows the average values of these metrics across all the time windows of the entire 9-month Webserver trace.

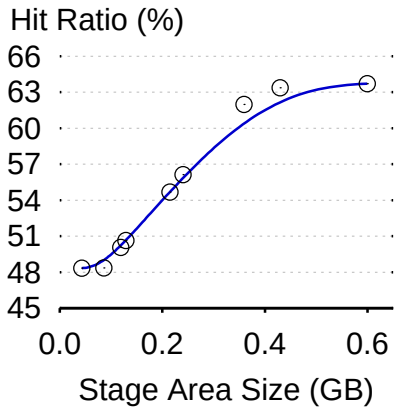
The results show that the difference in hit ratio is small among the different prediction methods but is considerable in cache allocation. The last value method has the highest prediction error, which confirms the need of prediction techniques. The exponential smoothing methods have the lowest prediction errors, and *Exp Self*

is more preferable because it automatically trains its parameter. We believe that more advanced prediction methods are possible to further improve the prediction accuracy and our solution can be extended to run multiple prediction methods at the same time and choose the best one at runtime. But this simple smoothing-based method can already produce good results, as shown in the following experiments which all use *Exp Self* to predict cache demand.

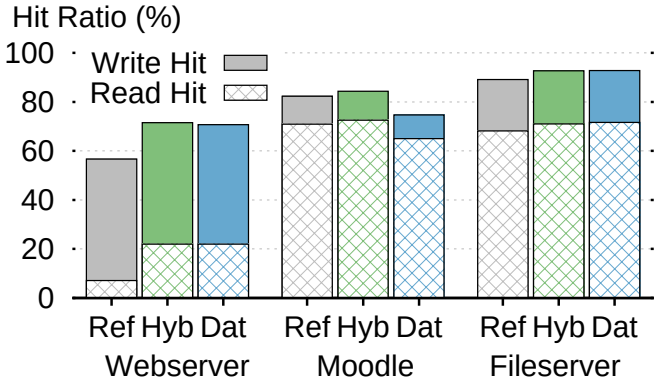
Staging Strategies

In the second set of experiments, we evaluate CloudCache's staging strategies. First, we study the impact of the staging area size. In general, it should be decided according to the number of VMs consolidated to the same cache and the IO intensity of their workloads. Therefore, our approach is to set the total staging area size as a percentage, e.g., between 0.1% and 1%, of the flash cache size, and allocate the staging area to the workloads proportionally to their flash cache allocation. Figure 3.6(a) gives an example of how the staging area allocation affects the Webserver workload's hit ratio when using address staging. The results from data staging are similar. In the rest of the chapter, we always use 256MB as the total staging area size for RWSS-based cache allocation. Note that we need 24B of the staging space for tracking each address, and an additional 4KB if its corresponding data is also staged.

Next we compare the address, data, and hybrid staging (with a 1:7 ratio between address and data staging space) strategies with the same staging area size in Figure 3.6(b). Data staging achieves a better read hit ratio than address staging by 67% for the Webserver trace but it loses to address staging by 9% for Moodle. These results confirm our discussion in Section 3.4.3 about the tradeoff between these strategies. In comparison, the hybrid staging combines the benefits of these



(a) Staging area size



(b) Staging strategies

Figure 3.6: Staging strategy analysis

two and is consistently the best for all traces. We have tested different ratios for hybrid staging, and our results show that the hit ratio difference is small (<1%). But a larger address staging area tracks a longer history and admits more data into the cache, which results in more cache usage and flash writes. Therefore, in the rest of this chapter, we always use hybrid staging with 1:7 ratio between address and data staging space for RWSS-based allocation.

We also compare to the related work High Endurance Cache (HEC) [YPGT13] which used two cache admission techniques to address flash cache wear-out and are

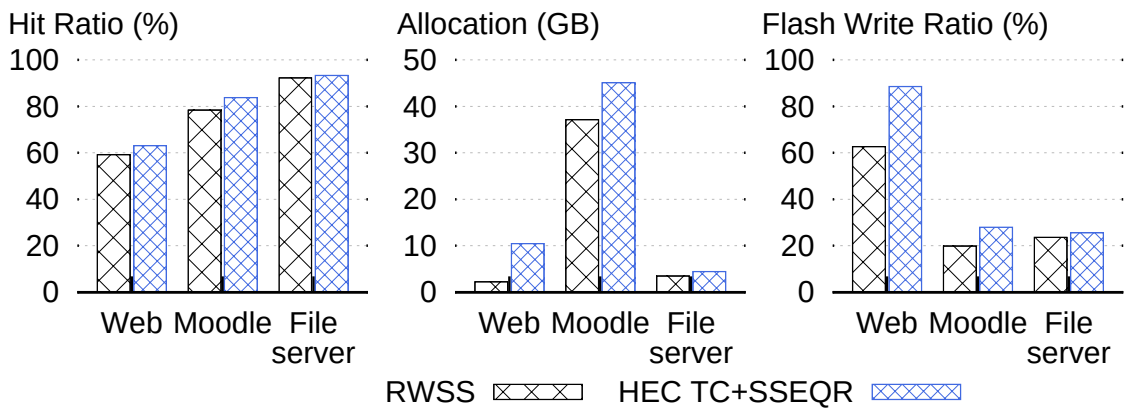


Figure 3.7: Comparison to HEC

closely related to our staging strategies. HEC's Touch Count (TC) technique uses an in-memory bitmap to track all the cache blocks (by default 4MB) and admit only reused blocks into cache. In comparison, CloudCache tracks only a small number of recently accessed addresses to limit the memory usage and prevent blocks accessed too long ago from being admitted into cache. HEC's Selective Sequential Rejection (SSEQR) technique tracks the sequentiality of accesses and rejects long sequences (by default any longer-than-4MB sequence). In comparison, CloudCache uses the staging area to automatically filter out long scan sequences.

Because HEC did not consider on-demand cache allocation, we implemented it by using TC to predict cache demand and using both TC and SSEQR to enforce cache admission. Figures 3.7 shows the comparison using the different traces, which reveals that on average HEC allocates up to 3.7x more cache than our RWSS-based method and causes up to 29.2% higher flash write ratio—the number of writes sent to cache device vs. the number of IOs received by cache. In return, it achieves only up to 6.4% higher hit ratio. The larger cache allocation in HEC is because it considers all the historical accesses when counting reuses, whereas the RWSS method

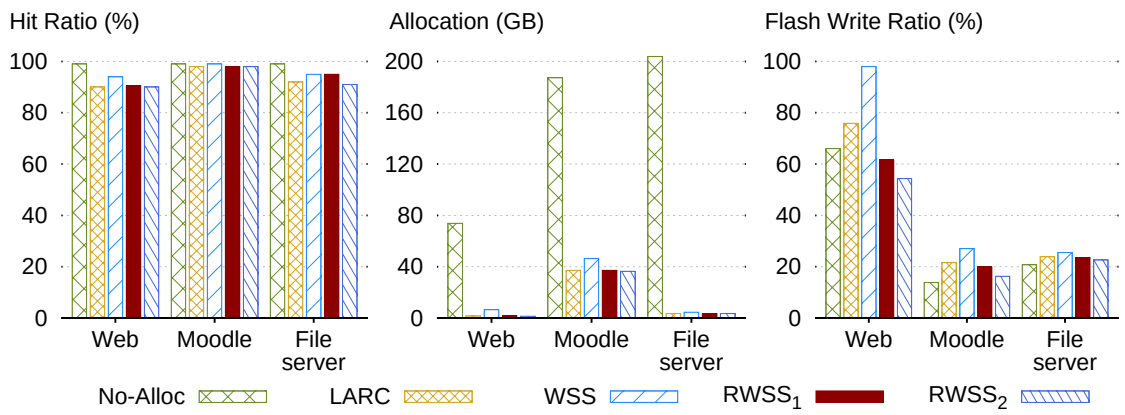
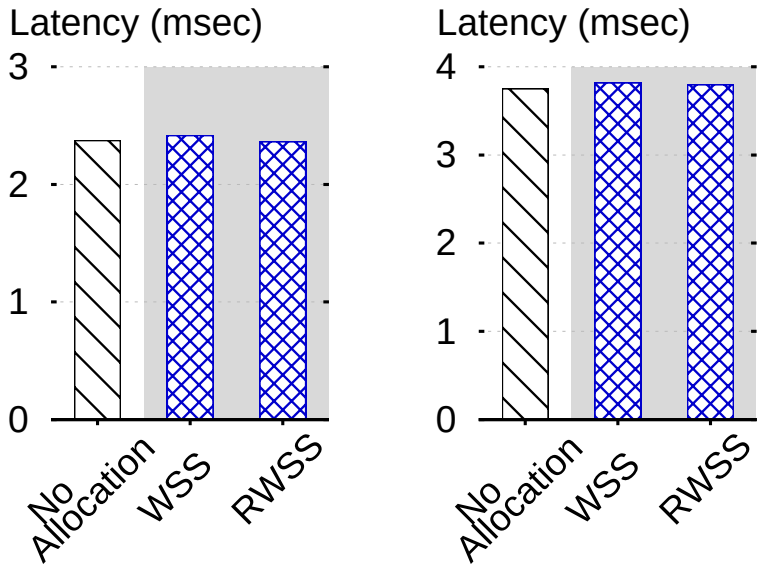


Figure 3.8: Allocation methods

considers only the reuses occurred in the recent history—the previous window. (If we were able to apply the same cache allocation given by the RWSS method while using HEC's cache admission method, we would achieve a much lower hit ratio, e.g., 68% lower for Moodle, and still a higher flash write ratio, e.g., 69% higher for Moodle.) The result also confirms that the RWSS method is able to automatically reject scan sequences (e.g., it rejects on average 90% of the IOs during the backup periods), whereas HEC needs to explicitly detect scan sequences using a fixed threshold.

WSS vs. RWSS-based Cache Allocation

In the third set of experiments, we compare RWSS-based to WSS-based cache allocation using the same prediction method, exponential smoothing with self-tuning. In both cases, the cache allocation is strictly enforced, and at the start of each window, the workload's extra cache usage beyond its new allocation is immediately dropped. This setting produces the *worst-case* result for on-demand cache allocation, because in practice CloudCache allows a workload to use spare capacity beyond its allocation and its extra cache usage is gradually reclaimed via replacement-time



(a) Webserver (b) Moodle

Figure 3.9: VM IO latency comparison

enforcement. We also include the case where the workload can use up the entire cache as a baseline (*No Allocation*), where the cache is large enough to hold the entire working set and does not require any replacement.

Figure 3.8 shows the comparison among these different approaches. For RWSS, we consider two different values for the N in RWS_N, as described in Section 3.4.1. In addition, we also compare to the related cache admission method, LARC [HWC+13], which dynamically changes the size of the staging area according to the current hit ratio—a higher hit ratio reduces the staging area size. Like HEC, LARC also does not provide on-demand allocation, so we implemented it by using the number of reused addresses to predict cache demand and using LARC for cache admission.

$RWSS_1$ achieves a hit ratio that is only 9.1% lower than *No Allocation* and 4% lower than *WSS*, but reduces the workload's cache usage substantially by up to 98% compared to *No Allocation* and 76% compared to *WSS*, and reduces the flash write

ratio by up to 6% compared to *No Allocation* and 37% compared to *WSS*. (The cache allocation of RWSS and LARC is less than 4GB for Webserver and Fileserver and thus barely visible in the figure). *No Allocation* has slightly lower flash write ratio than $RWSS_1$ for Moodle and Fileserver only because it does not incur cache replacement, as it is allowed to occupy as much cache space as possible, which is not a realistic scenario for cloud environments. Compared to *LARC*, $RWSS_1$ achieves up to 3% higher hit ratio and still reduces cache usage by up to 3% and the flash writes by up to 18%, while using 580MB less staging area on average. Comparing the two different configurations of RWSS, $RWSS_2$ reduces cache usage by up to 9% and flash writes by up to 18%, at the cost of 4% lower hit ratio, which confirms the tradeoff of choosing different values of N in our proposed RWS model.

To evaluate how much performance loss the hit ratio reduction will cause, we replay the traces and measure their IO latencies. We consider a one-month portion of the Webserver and Moodle traces. They were replayed on the real VM storage and caching setup specified in Section 3.4.4. We compare the different cache management methods in terms of 95th percentile IO latency. Figure 3.9 shows that the *RWSS*-based method delivers the similar performance as the alternatives (only 1% slower than *No Allocation* for Moodle) while using much less cache and causing more writes to the cache device as shown in the previous results.

The results confirm that our proposed RWSS-based cache allocation can indeed substantially reduce a workload's cache usage and the corresponding wear-out at only a small performance cost. In real usage scenarios our performance overhead would be much smaller because a workload's extra cache allocation does not have to be dropped immediately when a new time window starts and can still provide hits while being gradually replaced by the other workloads. Moreover, because the WSS-based method requires much higher cache allocations for the same workloads, cloud

providers have to either provision much larger caches, which incurs more monetary cost, or leave the caches oversubscribed, which leads to bad performance as the low-locality data are admitted into the cache and flush out the useful data.

3.5 Summary

Flash caching has great potential to address the storage bottleneck and improve VM performance for cloud computing systems. Allocating the limited cache capacity to concurrent VMs according to their demands is key to making efficient use of flash cache and optimizing VM performance. Moreover, flash devices have serious endurance issues, whereas weak-temporal-locality data are abundant at the flash cache layer, which hurt not only the cache performance but also its lifetime. Therefore, on-demand management of flash caches requires fundamental rethinking how to estimate VMs' cache demands and how to provision space to meet their demands.

This chapter presents CloudCache, an on-demand cache management solution to these problems. It employs a new cache demand model, Reuse Working Set (RWS), to capture the data with good temporal locality, allocate cache space according to the predicted Reuse Working Set Size (RWSS), and admit only the RWS into the allocated space. Extensive evaluations based on real-world traces confirm that the RWSS-based cache allocation approach can achieve good cache hit ratio and IO latency for a VM while substantially reducing its cache usage and flash wear-out.

CHAPTER 4

Dynamic Cache Migration

4.1 Introduction

The *on-demand cache allocation* approach discussed in the previous chapter allows CloudCache to estimate the cache demands of workloads online and dynamically allocate the shared capacity to them. Using this approach CloudCache is able to reduce cache usage and wear-out, but in order to meet workloads demands the cache capacity has to be sufficient to hold all the current VMs. This chapter address another key question when using on-demand cache allocation, which is *how to handle situations where the VMs' cache demands exceed the flash cache's capacity.* Due to the dynamic nature of cloud workloads, such cache overload situations are bound to happen in practice and VMs will not be able to get their desired cache capacity. To solve this problem, we propose a *dynamic cache migration* approach to balance cache load across hosts by live migrating the cached data along with the VMs. It uses both on-demand migration of dirty data to provide zero downtime to the migrating VM, and background migration of RWS to quickly warmup the cache for the VM, thereby minimizing its performance impact. Meanwhile, it can also limit the data transfer rate for cache migration to limit the impact to other co-hosted VMs.

Our results show that the proposed dynamic cache migration reduces the VM's IO latency by 93% compared to no cache migration, and causes at most 21% slow-down to the co-hosted VMs during the migration. Combining the on-demand cache allocation with the dynamic cache migration approaches, CloudCache is able to im-

prove the average hit ratio of 12 concurrent VMs by 28% and reduce their average 90th percentile IO latency by 27%, compared to the case without cache allocation.

The rest of the chapter is organized as follows: Section 4.2 describes live VM migration, Section 4.3 presents how to limit the transfer rate, Section 4.4 discusses our evaluation results and finally Section 4.5 presents and evaluation using 12 concurrent VMs.

4.2 Live Cache Migration

Live VM migration allows a workload to be transparently migrated among physical hosts while running in its VM [CFH+05, NLH05]. In CloudCache, we propose to use live VM migration to balance the load on the flash caches of VM hosts—when a host's cache capacity becomes insufficient to meet the local VMs' total cache demands (as estimated by their predicted RWSSes), some VMs can be migrated to other hosts that have spare cache capacity to meet their cache demands.

VM-migration-based cache load balancing presents two challenges. First, the migrating VM's dirty cache data on the migration *source* host must be synchronized to the *destination* host before they can be accessed again by the VM. A naive way is to flush all the dirty data to the remote storage server for the migrating VM. Depending on the amount of dirty data and the available IO bandwidth, the flushing can be time consuming, and the VM cannot resume its activity until the flushing finishes. The flushing will also cause a surge in the storage server's IO load and affect the performance of the other VMs sharing the server. Second, the migrating VM needs to warm up the cache on the destination host, which may also take a long time, and it will experience substantial performance degradation till the cache is warmed up [HAWS13, AZ14].

To address these challenges, CloudCache's dynamic cache migration approach uses a combination of reactive and proactive migration techniques:

On-Demand Migration: When the migrated VM accesses a block that is dirty in the source host's cache, its local cache forwards the request to the source host and fetches the data from there, instead of the remote storage server. The metadata of the dirty blocks, i.e., their logical block addresses, on the source host are transferred along with VM migration, so the destination host's local cache is aware of which blocks are dirty on the source host. Because the size of these metadata is small (e.g., 8B per 4KB data), the metadata transfer time is often negligible. It is done before the VM is activated on the destination, so the VM can immediately use the cache on the destination host.

Background Migration: In addition to reactively servicing requests from the migrated VM, the source host's cache also proactively transfers the VM's cached data—its RWS—to the destination host. The transfer is done in background to mitigate the impact to the other VMs on the source host. This background migration allows the destination host to quickly warm up its local cache and improve the performance of the migrated VM. It also allows the source host to quickly reduce its cache load and improve the performance of its remaining VMs. Benefiting from the RWSS-based cache allocation and admission, the data that need to be transferred in background contain only the VM's RWS which is much smaller than the WS, as shown in the previous section's results. Moreover, when transferring the RWS, the blocks are sent in the decreasing order of their recency so the data that are most likely to used next are transferred earliest.

On-demand migration allows the migrated VM to access its dirty blocks quickly, but it is inefficient for transferring many blocks. Background migration can transfer bulk data efficiently but it may not be able to serve the current requests that

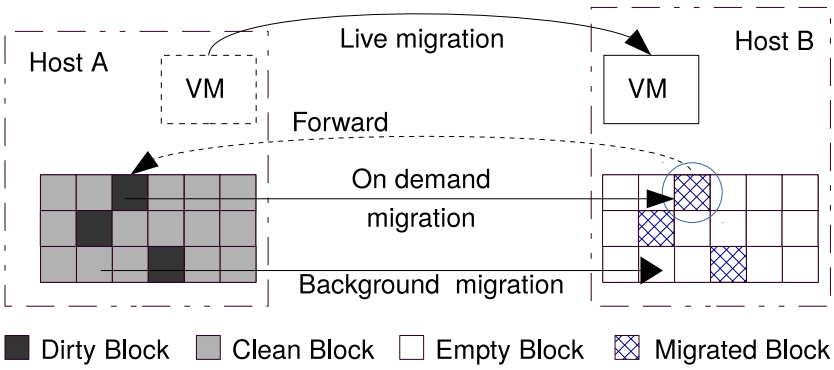


Figure 4.1: Architecture of dynamic cache migration

the migrated VM is waiting for. Therefore, the combination of these two migration strategies can optimize the performance of the VM. Figure 4.1 illustrates how CloudCache performs cache migration. When a VM is live-migrated from Host A to Host B, to keep data consistent while avoiding the need to flush dirty data, the cached metadata of dirty blocks are transferred to Host B. Once the VM live migration completes, the VM is activated on Host B and its local flash cache can immediately service its requests. By using the transferred metadata, the cache on Host B can determine whether a block is dirty or not and where it is currently located. If a dirty block is still on Host A, a request is sent to fetch it on demand. At the same time, Host A also sends the RWS of the migrated VM in background. As the cached blocks are moved from Host A to Host B, either on-demand or in background, Host A vacates their cache space and makes it available to the other VMs.

The CloudCache module on each host handles both the operations of local cache and the operations of cache migration. It employs a multithreaded design to handle these different operations with good concurrency. Synchronization among the threads is needed to ensure consistency of data. In particular, when the destination host requests a block on demand, it is possible that the source host also transfers this block in background, at the same time. The destination host will discard the second

copy that it receives, because it already has a copy in the local cache and it may have already overwritten it. As an optimization, a write that aligns to the cache block boundaries can be stored directly in the destination host's cache, without fetching its previous copy from the source host. In this case, the later migrated copy of this block is also discarded. The migrating VM needs to keep the same device name for its disk, which is the virtual block device presented by CloudCache's block-level virtualization. CloudCache assigns unique names to the virtual block devices based on the unique IDs of the VMs in the cloud system. Before migration, the mapping from the virtual block device to physical device (e.g., the iSCSI device) is created on the destination host, and after migration, the counterpart on the source host is removed.

4.3 Migration Rate Limiting

While the combination of on-demand and background migrations can optimize the performance of a migrating VM, the impact to the other VMs on the source and destination hosts also needs to be considered. Cache migration requires reads on the source host's cache and writes to the destination host's cache, which can slow down the cache IOs from the other co-hosted VMs. It also requires network bandwidth, in addition to the bandwidth already consumed by VM memory migration (part of the live VM migration [CFH+05, NLH05]), and affects the network IO performance of the other VMs.

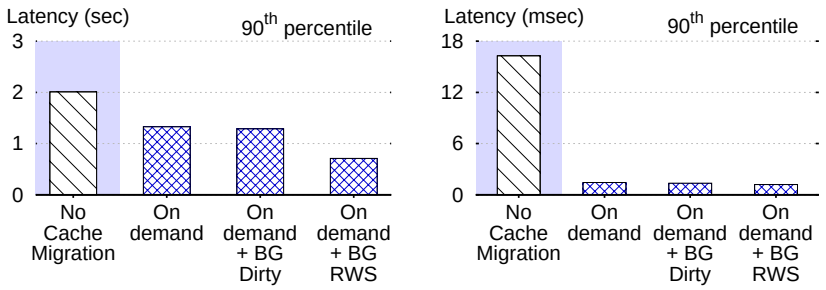
In order to control the level of performance interference to co-hosted VMs, CloudCache is able to limit the transfer rate for cache migration. Given the rate limit, it enforces the maximum number of data blocks that can be transferred from the source host to the destination host every period of time (e.g., 100ms), including both

on-demand migration and background migration. Once the limit is hit, the migration thread will sleep and wait till the next period to continue the data transfer. If on-demand requests arrive during the sleep time, they will be delayed and served immediately after the thread wakes up. The rate can be set based on factors including the priority of the VMs and the RWSS of the migrating VM. CloudCache allows a system administrator to tune the rate in order to minimize the cache migration impact to the co-hosted VMs and still migrate the RWS fast enough to satisfy the cache demands.

4.4 Evaluation

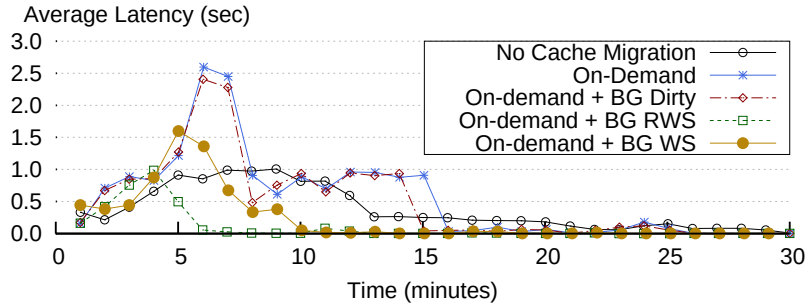
We evaluate the performance of CloudCache's dynamic cache migration using the same testbed described in Section 3.4.4. Dynamic cache migration is implemented in the CloudCache kernel module described in Section 3.4.4. It exposes a command-line interface which is integrated with virt-manager [vir] for coordinating VM migration with cache migration. We focus on a day-long portion of the Moodle and Webserver traces. The Moodle one-day trace is read-intensive which makes 15% of its cached data dirty (about 5GB), and the Webserver one-day trace is write-intensive which makes 85% of its cached data dirty (about 1GB).

We consider four different approaches: (1) *No Cache Migration:* the cached data on the source host are not migrated with the VM; (2) *On-demand:* only the on-demand cache migration is used to transfer dirty blocks requested by the migrated VM; (3) *On-demand + BG Dirty:* in addition to on-demand cache migration, background migration is used to transfer only the dirty blocks of the migrated VM; (4) *On-demand + BG RWS:* both on-demand migration of dirty blocks and background migration of RWS are used. In this experiment, we assume that the cache migration



(a) Moodle

(b) Webserver



(c) The migrating VM's performance (average IO latency per minute) for Moodle. The migration starts at the 5th minute.

Figure 4.2: Migration strategies

can use the entire 1Gbps network bandwidth, and we study the impact of rate limiting in the next experiment. For on-demand cache migration, it takes 0.3s to transfer the metadata for the Moodle workload and 0.05s for the Webserver workload.

Figure 4.2(a) shows that for the Moodle workload, on-demand cache migration decreases the 90th percentile latency by 33% and the addition of background migration of dirty data decreases it by 35%, compared to *No Cache Migration*. However, the most significant improvement comes from the use of both on-demand migration of dirty data and background migration of the entire RWS, which reduces the latency by 64%. The reason is that this workload is read-intensive and reuses a large amount of clean data; background migration of RWS allows the workload to access these data from the fast, local flash cache, instead of paying the long network latency for accessing the remote storage.

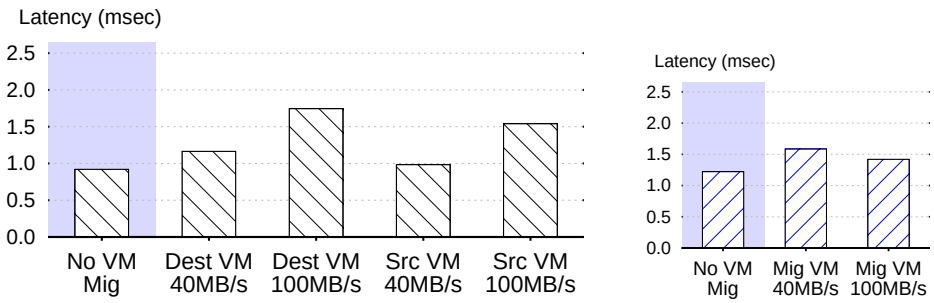
For the Webserver workload, because its RWS is mostly dirty, the difference among the three cache migration strategies is smaller than the Moodle workload (Figure 4.2(b)). Compared to the *No Cache Migration* case, they reduce the 90th percentile latency by 91.1% with on-demand migration of dirty data, and by 92.6% with the addition of background migration of RWS.

Note that the above results for the *No Cache Migration* case do not include the time that the migrated VM has to wait for its dirty data to be flushed from the source host to the remote storage before it can resume running again, which is about 54 seconds for the Moodle workload and 12 seconds for the Webserver workload, assuming it can use all the bandwidths of the network and storage server. In comparison, the VM has zero downtime when using our dynamic cache migration.

Figure 4.2(c) shows how the migrating VM's performance varies over time in this Moodle experiment so we can observe the real-time performance of the different migration strategies. The peaks in *On-demand* and *On-demand + BG Dirty* are caused by bursts of on-demand transfer of clean data blocks requested by the migrated VM. We believe that we can further optimize our prototype implementation to avoid such spikes in latency.

In Figure 4.2(c), we also compare our approach to an alternative cache migration implementation (*On-demand + BG WS*) which migrates the VM's entire working set without the benefit of our proposed RWS model. Using the same Moodle trace, at the time of migration, its RWSS is 32GB and WSS is 42GB. As a result, migrating the WS takes twice the time of migrating only the RWS (6mins vs. 3mins) and causes a higher IO latency overhead too (71% higher in 90th percentile latency).

In the next experiment, we evaluate the performance impact of rate limiting the cache migration. In addition to the migrating VM, we run another IO-intensive VM on both the source and destination hosts, which replays a different day-long portion



(a) Co-hosted VMs (b) Migrating VM

Figure 4.3: Impact of different cache migration rate

of the Webserver trace. We measure the performance of all the VMs when the cache migration rate is set at 40MB/s and 100MB/s and compare to their normal performance when there is no VM or cache migration. Figure 4.3 shows that the impact to the co-hosted VMs' 90th percentile IO latency is below 16% and 21% for the 40MB/s and 100MB/s rate respectively. Note that this is assuming that the co-hosted VMs already have enough cache space, so in reality, their performance would actually be much improved by using the cache space vacated from the migrating VM. Meanwhile, the faster migration rate reduces the migrating VM's 90th percentile IO latency by 6%. Therefore, the lower rate is good enough for the migrating VM because the most recently used data are migrated first, and it is more preferable for its lower impact to the co-hosted VMs.

4.5 Putting Everything Together

The previous two sections and Chapter 3 described and evaluated the RWSS-based on-demand cache allocation and dynamic cache migration approaches separately. In this section, we present how to use them together to realize on-demand cache management for multiple VM hosts. Consider the flash cache on a single host. If its

capacity is sufficient to satisfy the predicted cache demands for all the local VMs, it is simply allocated to the VMs according to their demands. The spare capacity is distributed to the VMs proportionally to their demands, or left idle to minimize wear-out. If the cache capacity is not sufficient, then cache migration needs to be considered in order to satisfy the demands of all the VMs.

When considering the use of cache migration, there are three key questions that need to be answered, *when to migrate*, *which VM to migrate*, and *which host to migrate it to?* To answer the first question, CloudCache reserves a certain percentage (e.g., 10%) of the cache capacity as a buffer to absorb the occasional surges in cache demands, and it starts a migration when the total cache demand exceeds the 90% threshold for several consecutive RWSS windows (e.g., three times). This approach prevents the fluctuations in cache workloads from triggering unnecessary cache migrations which affect the VMs' performance and the system's stability.

To answer the second and third questions, CloudCache's current strategy is to minimize the imbalance of cache load among the hosts in the system. The host that requires cache migration queries every other host's current cache load. It then evaluates all the possible migration plans of moving one of its local VMs to a host that can accommodate the VM's RWS under the 90% threshold. It then chooses the plan that minimizes the variance of the hosts' cache load distribution.

We use a real experiment to illustrate the use of our approaches for meeting the cache demands of dynamically changing workloads. We consider two VM hosts each with 64GB of flash cache. Host A ran 12 VMs, and Host B ran three VMs, concurrently. Each VM replayed a different 10-day portion of the Webserver trace. The cache allocation was adjusted every 2 days on both hosts. The first time window is the warm-up phase during which the VMs were given equal allocation of the cache capacity. Afterwards, the cache was allocated to the VMs proportionally to their

estimated RWSSes. Moreover, a VM could take more than its share if there was idle capacity from the other VMs' shares because our approach is work-conserving. The experiment was done on the real VM storage and caching setup specified in Section 3.4.4.

Figure 4.4(a) shows how the cache space is distributed among the VMs on Host A when *(a)* there is no cache allocation, *(b)* on-demand cache allocation but without cache migration, and *(c)* on-demand cache allocation with dynamic cache migration. Comparing (a) and (b), we can see how our RWSS-based on-demand allocation improves the fairness among the competing VMs. For example, between Days 4 and 8, VMs 6, 7, 8 dominated the cache space in (a), but in (b), every VM got a fair share of the cache space proportionally to their estimated RWSSes. Notice that VMs 7 and 8 were allocated much less in (b) than what they got in (a), which is an evidence of how the RWS-based cache demand model filtered out the VMs' low-locality data and kept only those that are useful to their performance. As a result, comparing the average performance of all 12 VMs across the entire experiment, (b) is better than (a) by 17% in terms of hit ratio and 13% in terms of 90th percentile IO latency.

In (c) dynamic cache migration was enabled in addition to on-demand cache allocation. After the total demand—the sum of the 12 VMs' RWSSes—exceeded the threshold for three consecutive windows, CloudCache initiated cache migration on Day 8 and chose to move VM 11, the one with the largest predicted RWSS at that time, and its cached data to Host *B*. As VM 11's RWS was moved to Host *B*, the remaining 11 VMs took over the whole cache on Host *A*, proportionally to their estimated RWSSes. As a result, comparing the average performance of all 12 VMs after Day 8, (c) is better than (b) by 49% in terms of hit ratio and 24% in terms of 90th percentile IO latency. Across the entire experiment, it outperforms (a) by 28%

in hit ratio and 27% in 90th percentile IO latency, and outperforms (b) by 10% in hit ratio and 16% in 90th percentile IO latency.

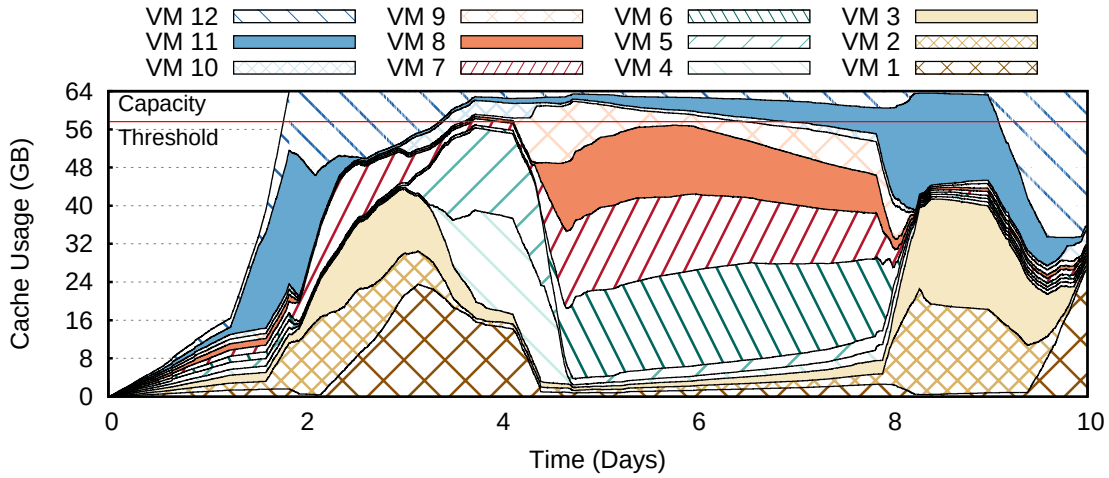
Although this experiment involved only two VM hosts and the migration of only one VM, the above results are still representative for the migration of any VM and its cache data between two hosts in a large cloud computing environment. But we understand in such a large environment, more intelligence is required to make the optimal VM migration decisions. There is a good amount of related work (e.g., [WSVY07, XF11]) on using VM migration to balance load on CPUs and main memory and to optimize performance, energy consumption, etc. CloudCache is the first to consider on-demand flash cache management across multiple hosts, and it can be well integrated into these related solutions to support the holistic management of different resources and optimization for various objectives. We leave this to our future work because the focus of this chapter is on the key mechanisms for on-demand cache management, i.e., on-demand cache allocation and dynamic cache migration, which are missing in existing flash cache management solutions and are non-trivial to accomplish.

4.6 Summary

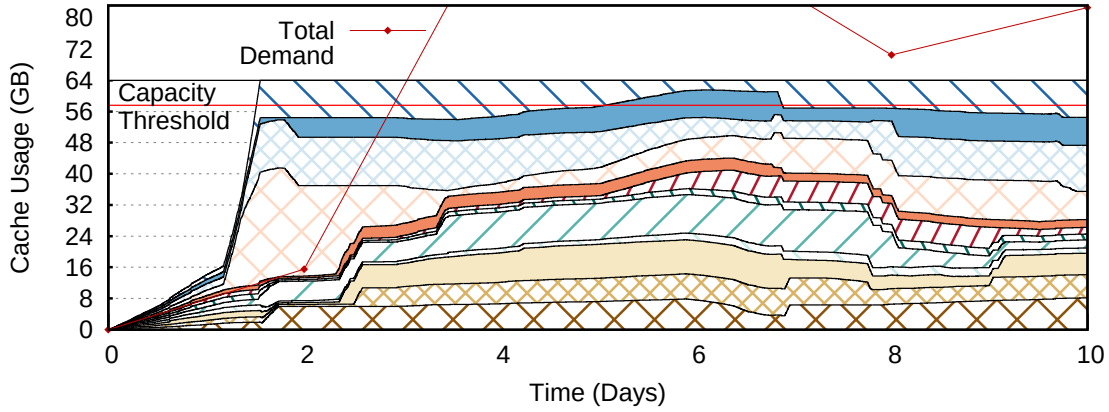
Flash caching has great potential to address the storage bottleneck of cloud computing systems and improve associated VM performance. Allocating limited cache capacity to concurrent VMs according to their demands is key to efficient use of flash cache and optimizing VM performance, as shown in the previous chapter. But this approach does not provide for the scenario in which the total demand of concurrent VMs is higher that the available cache capacity, a scenario that will become increas-

ingly common with the increasing level of consolidation on cloud systems. Therefore, cache allocation alone does not address the scalability issue of cloud storage.

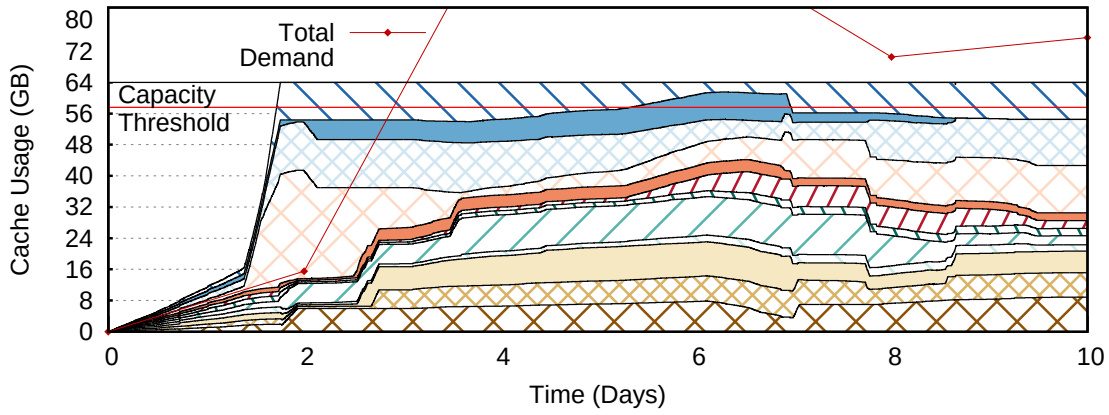
This chapter presents dynamic cache migration to handle cache overload situations. This approach live-migrates a VM with its cached data to meet the cache demands of all VMs, employing on-demand cache migration for dirty data and background migration of the entire reuse working set. Our results confirm that dynamic cache migration can transparently balance cache load across hosts with little impact on the migrating VM and the other cohosted VMs



(a) No cache allocation



(b) On-demand cache allocation without cache migration



(c) On-demand cache allocation with dynamic cache migration

Figure 4.4: Cache usages of 12 concurrent VMs

CHAPTER 5

Related Work

5.1 Flash Caching Solution

Client-side disk caching can improve the performance of cloud storage by harnessing the storage available on the client-side of network storage, the VM hosts, and the locality inherent in VM I/Os. With the emergence of solid state drives (SSDs), the benefit of client-side caching is potentially more significant as the speed of an SSD cache substantially outperforms the storage server that has traditional spinning disks.

The potential of flash caching has motivated several related solutions. For example: Mercury [BLM+12] is a persistent, write-through host-side cache for flash memory that was designed as a hypervisor cache, which simplifies its integration and deployment into host environments, in order to provide caching to the VMs hosted on the client over a variety of networked storage protocols; ioCache [ioC] supports caching in the hypervisor and in the individual VMs on a storage client using custom-built flash hardware and management software; HyCache [ZR12] is a distributed middle-ware layer built on top of HDFS. It creates a user level cache API with SSD device to speed up I/O workloads on HDFS, where each distributed data node has a HDD and SSD where a subset of HDFS block files are kept on SSD for performance enhancement. Strong consistency is made possible by creating a symbolic link from SSD to HDD after eviction. In this thesis, we base our flash caching study on *dm-cache* [dmc], a open-source block-level caching solution. It is created upon block-device virtualization and can be transparently deployed on

VM hosts. It has been successfully adopted by production cloud systems [dmc] and motivated the designs of other related solutions (e.g., FlashCache [fla]).

There are also related works on improving various aspects of SSD caching. For example, FlashTier [SSZ12] proposed cache-specific SSD management to enhance the performance of an SSD device dedicated for caching uses, it provides unified address space by using sparse hash map from Google, along with cache consistency and silent eviction for improved performance; Previous work from Koller *et al.* also advocated the importance of write-back caching and studied new ordered and journaled write-back policies for flash caches, in order to improve the consistency of cached dirty data [KMR+13]. This thesis complements the previous work by further studying the performance impact of write-back caching to both storage client and server using real workloads and proposing a new cache-optimized RAID technique to improve the reliability of write-back-based flash caches.

Cache configuration has also been studied by Holland et al. [HAWS13], which focuses on several key design considerations, including flash-RAM integration, write-back policy, cache persistency, and cache consistency. Their study is based on simulations that calculates the latencies generated by different write policy. Their conclusion is that write-through caching delivers good enough performance compared to write-back since writes can be asynchronously submitted to the back-end storage. However, this work does not consider the impact on the servers load and its resulting effect on the clients performance, which are studied in this thesis using a real flash cache implementation analyzing traces collected from highly consolidated cloud environments.

RAID is a classic technique to improve the reliability of storage and has also been considered in the context of flash storage [JMBR11]. A unique challenge of flash-based RAID is synchronous aging, which means that the flash devices used

in a RAID group wear out at the same time and cannot be recovered by RAID. Diff-RAID was proposed to address this challenge by intentionally distributing parity blocks unevenly across the flash devices so that the writes caused by parity updates are also unevenly distributed, allowing the devices to wear out at different speed [BKPM10]. This related work is complementary to the proposed cache-optimized RAID technique which improves storage utilization and reduces wear out by providing redundancy to only the dirty data in a cache.

5.2 Cache Management

There are several related flash cache management solutions. S-CAVE [LML+13] considers the number of reused blocks to estimate a VM's cache demands, and it does not employ a cache admission policy. Hence, the non-reused blocks also require cache space and may evict the more useful blocks. Moreover, the cache allocation in S-CAVE is done using several heuristics. vCacheShare [MZM+14] allocates a read-only cache by maximizing a unity function that captures a VM's disk latency, read-to-write ratio, estimated cache hit ratio, and reuse rate of the allocated cache capacity. However, these solutions do not allocate cache capacity according to the VMs' actual cache demands, nor do they consider dynamic cache migration for meeting the demands when a cache becomes overloaded. These problems are addressed by CloudCache's on-demand cache allocation and dynamic cache migration approaches.

HEC and LARC studied cache admission policies to filter out data with weak temporal locality and reduce the flash wear-out [YPGT13, HWC+13, LCQX14]. However, they do not consider the problem of how to allocate shared cache capacity to concurrent workloads, which is addressed by CloudCache. Moreover, our RWSS-

based approach is also able to effectively filter out data with no reuses and achieve good reduction on flash wear-out. Another similar approach is mARC [SLK+15] which also filters out data with weak temporal locality and does not consider allocation of shared cache capacity, it uses multiple phases which allow to admit more data than other approaches, based on implementation details we believe that our RWSS-based approach can achieve better reduction on flash wear-out.

In the context of processor and memory cache management, there are cache replacement algorithms that address the cache pollution caused by scan sequences. In particular, ARC keeps data with no reuses in a separate list and prevents it from flooding the list of data with reuses [MM03]. However, it does not address the wear-out caused by scan sequences, as data with reuses are still admitted into the cache. There are also related works on processor and memory cache allocations. For example, miss-rate curve (MRC) can be built to capture the relationship between a workload's cache hit ratio and its cache sizes. The cache allocation can be then decided by optimizing the overall performance of all the workloads [SRD01, ZPS+04, TASS09]. Related work has also studied the use of process migration to balance the process cache load on a multicore system [KBH+08].

Compared to processor and memory cache management, flash cache management has fundamentally different challenges and opportunities. On one hand, the workload at the flash cache layer can be highly spiky and contain long scan sequences which are detrimental to both the performance and endurance of the cache. On the other hand, flash cache management can employ more sophisticated techniques implemented in software and use VM migration to dynamically balance cache load across hosts. We exploit these opportunities and addresses these challenges in our on-demand flash cache management solution, CloudCache.

5.3 Cache Migration

Bhagwat et al. studied how to allow a migrated VM to request data from the cache on its pervious host [BPO+15], in the same fashion as the on-demand cache migration proposed in Section 4.1. However, as shown in our evaluation results, this technique alone cannot ensure good performance to the migrated VM. It also has a long-lasting, negative impact on the source host in terms of both performance interference and cache utilization. If the migrated VM's data are evicted by the source host, then its performance will be even worse because a request has to be forwarded by the source host to the primary storage. In comparison, CloudCache considers the combination of on-demand migration and background migration and optimizes the performance of both the migrated VM and the other co-hosted VMs.

CHAPTER 6

Conclusions and Future work

6.1 Conclusions

Caching is one of the most widely used techniques for improving the performance of data access in computer systems. Its effectiveness is largely determined by the available locality in the workload that can be exploited by the cache, and the speedup that can be obtained by serving it from the cache versus from the next layer in the storage hierarchy. The emergence of flash storage has motivated consideration of client-side caching in network storage systems because flash speed is significantly faster than that of the network and the mechanical disks on the storage server. It also comes in time to address the serious scalability issues that cloud computing systems now face as the number and size of VMs quickly increase on shared storage systems. Allocating limited cache capacity to concurrent VMs according to their demands is key to the efficient use of flash cache and to optimizing VM performance. Moreover, flash devices have serious endurance issues, whereas weak-temporal-locality data are abundant at the flash cache layer, hurting both cache performance and cache lifetime. Therefore, on-demand management of flash caches requires a fundamental rethinking of how to estimate VM cache demands and how to provision space to meet these demands.

This thesis presents three research components that address previous problems for using flash caching on cloud computing systems. First, it presents a thorough study of flash cache architecture and configurations, by analyzing a large number of real-world traces collected from both public and private clouds. This study confirms

that cloud workloads have good cacheability and dm-cache incurs low overhead with respect to commodity flash devices. The impact of different write caching policies is significant to cache performance. In contrast to conclusions reached in related studies, our results show that write-back caching can significantly outperform write-through caching due to the reduction of server I/O load. Our results also show the tradeoff of making a flash cache-persistent across client restarts, saving hours of cache warm-up time but incurring considerable overhead from persistent metadata updates. Finally, to address the reliability issue of write-back caching, we propose a new cache-optimized RAID technique that minimizes RAID overhead by introducing redundancy only to cached dirty data, which appears to be significantly faster than traditional RAID and write-through caching. Second, this thesis presents CloudCache, an on-demand cache management solution that employs a new cache demand model, RWS, to capture data with good temporal locality, allocate cache space according to the predicted RWSS, and admit only the RWS into the allocated space. Third, to handle cache overload situations, CloudCache takes a new cache migration approach that live-migrates a VM with its cached data to meet the cache demands of the VMs. Extensive evaluation based on real-world traces confirms that RWSS-based cache allocation can achieve a strong cache-hit ratio and I/O latency for a VM while significantly reducing its cache usage and flash wear. It also confirms that dynamic cache migration can transparently balance cache load across hosts with little impact on the migrating VM and the other cohosted VMs.

6.2 Future Work

This thesis provides a solid framework for future work in several directions. First, we plan to use flash simulators and open-controller devices to monitor the actual Pro-

gram/Erase cycles and provide more accurate measurement of our solution's impact on flash device wear-out. Second, when the aggregate cache capacity of all VM hosts is insufficient, CloudCache has to allocate cache proportionally to the VMs' RWSSes. We plan to investigate a more advanced solution that maps each VM's cache allocation to its performance and optimizes the allocation by maximizing the overall performance of all VMs, we plan to extend traditional miss-rate curve techniques [WPGA15, WID+14] to capture the relationship between cache miss rate and cache allocation. Third, although our experiments confirm that flash cache allocation has a significant impact on application performance, the allocation of other resources, e.g., CPU cycles and memory capacity, is also important. We plan to consider the holistic management of different resources including processor, memory, and flash cache and optimize the management for various objectives. Finally, although the discussion in this thesis focuses on flash-memory-based caching, CloudCaches general approach also applies to emerging nonvolatile memory (NVM) devices such as PCM and 3D XPoint used for caching in the storage hierarchy. These devices may or may not have the same endurance issue as NAND flash, but their capacities would still be quite limited and thus require an effective on-demand cache management solution. Nonetheless, cache management must be fast enough to match their much higher speeds. We plan to investigate the effectiveness of CloudCache for managing new NVM-based caches as they become available.

BIBLIOGRAPHY

[AZ14] Dulcardo Arteaga and Ming Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR 14)*, pages 7:1–7:11. ACM, 2014. URL: `http://doi.acm.org/10.1145/2611354.2611372`, `doi:10.1145/2611354.2611372`.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*. ACM, 2003.

[BKPM10] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):4, 2010.

[blk] *blktrace: Linuz block I/O traces.* `http://linux.die.net/man/8/blktrace`.

[BLM+12] Steve Byan, James Lentini, Anshul Madan, Luis Pabon, Michael Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, and Mark Storer. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST 12)*, Pacific Grove, CA, USA, 2012. IEEE.

[BPO+15] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 287–300, Santa Clara, CA, 2015. USENIX Association.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 273–286. USENIX Association, 2005.

[cloa] Cloud VPS. `https://www.cloudvps.nl/`.

[clob] Cloud VPS Activates Linux SSD Caching with dm-cache. http://www.cloudvps.com/blog/cloudvps-activates-linux-ssd-caching-with-dm-cache.

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[dmc] Dynamic block-level storage caching for cloud computing systems. http://visa.cs.asu.edu/tiki/dm-cache.

[dtr] *Dtrace: dynamic tracing framework by Sun Microsystems*. http://en.wikipedia.org/wiki/DTrace.

[ebs] Amazon Elastic Block Store. http://aws.amazon.com/ebs/.

[fio] *Fio - Flexible I/O Tester Synthetic Benchmark*. http://git.kernel.dk/?p=fio.git.

[fla] Facebook Flashcache. https://github.com/facebook/flashcache/.

[glo] GFS Project Page. http://sourceware.org/cluster/gfs/.

[HAWS13] David A Holland, Elaine Lee Angelino, Gideon Wald, and Margo I Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC 13)*. USENIX Association, 2013.

[HP06] John L. Hennessy and David A. Patterson. *Computer architecture - a quantitative approach, 4th Edition*. Morgan Kaufmann, 2006.

[HWC+13] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, pages 1–10. IEEE, 2013.

[HZ06] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report RC24123, IBM, November 2006.

[ioC] Fusion-io ioCache. http://www.fusionio.com/products/iocache/.

[JMBR11] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. The pitfalls of deploying solid-state drive raids. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 14. ACM, 2011.

[KBH+08] Rob C. Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008. URL: `http://doi.ieeecomputersociety.org/10.1109/MM.2008.48`.

[KHSB02] Marjorie Krueger, Randy Haagens, Costa Sapuntzakis, and Mark Bakke. Small computer systems interface protocol over the internet (iSCSI): Requirements and design considerations. Internet RFC 3347, July 2002.

[KMR+13] Ricardo Koller, Leonardo Marmol, Raju Ranganswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST 13)*, 2013.

[kvm] Kernel Based Virtual Machine. `http://www.linux-kvm.org/page/Main_Page`.

[LCQX14] Jian Liu, Yunpeng Chai, Xiao Qin, and Yuan Xiao. Plc-cache: Endurable ssd cache for deduplication-based primary storage. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–12, June 2014.

[LML+13] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT 13)*, pages 103–112. IEEE Press, 2013.

[LPGM08] Andrew Leung, Shankar Pasupathy, Garth Goodson, and Ethan Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of USENIX ATC*, 2008.

[LSD+14] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 501–512. USENIX Association, 2014.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1090694.1090708`.

[msr] MSR cambridge traces. `http://iotta.snia.org/traces/388`.

[MZM+14] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*, pages 133–144, Philadelphia, PA, June 2014. USENIX Association. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/meng`.

[nbd] Network Block Device. `http://nbd.sourceforge.net/`.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC 05)*, pages 391–394. USENIX, 2005.

[nov] Openstack Compute Documentation. `http://nova.openstack.org/index.html`.

[NWO88] Michael N Nelson, Brent B Welch, and John K Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[SLK+15] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To arc or not to arc. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, July 2015. USENIX Association. URL: `https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/santana`.

[SRD01] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings*

of the IASTED International Conference on Parallel and Distributed Computing and Systems (ICPADS 01), pages 116–127, 2001.

[SSZ12] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2168836.2168863`, `doi:10.1145/2168836.2168863`.

[TASS09] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*, pages 121–132, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1508244.1508259`, `doi:10.1145/1508244.1508259`.

[TS00] R.H. Thornburgh and B. Schoenborn. *Storage Area Networks*. Prentice Hall PTR, 2000.

[vir] Manage virtual machines with virt-manager. `https://virt-manager.org`.

[VMF] VMware VMFS. `http://www.vmware.com/products/vmfs/overview.html`.

[WID+14] Jake Wires, Stephen Ingram, ZacWarfield. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, October 2014. USENIX Association. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires`.

[WPGA15] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger`.

[WSVY07] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migra-

tion. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI 07)*, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1973430.1973447.

[XF11] Jing Xu and José Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC 11)*, pages 225–234, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1998582.1998636, doi:10.1145/1998582.1998636.

[YPGT13] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. HEC: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR 13)*, page 10. ACM, 2013.

[ZF06] Ming Zhao and R. J. Figueiredo. Application-tailored cache consistency for wide-area file systems. In *Proc. Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 41–41, 2006.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 04)*, pages 177–188, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1024393.1024415, doi:10.1145/1024393.1024415.

[ZR12] Dongfang Zhao and Ioan Raicu. Hycache: A hybrid user-level file system with ssd caching. In *1st Greater Chicago Area System Research Workshop*, 2012.

[ZZF06] Ming Zhao, Jian Zhang, and Renato Figueiredo. Distributed file system virtualization techniques supporting on-demand virtual machine environments for grid computing. *Cluster Computing*, 9(1):45–56, January 2006. doi:10.1007/s10586-006-4896-x.

VITA

DULCARDO A. ARTEAGA CLAVIJO

February 22, 1985	Born, Santa Cruz, Bolivia
2009 - 2016	Research Assistant Florida International University Miami, Florida
2012	M.S., Computer Science Florida International University Miami, Florida
2005	B.S., Computer Systems Engineering Universidad Mayor de San Simon Cochabamba, Bolivia

PUBLICATIONS AND PRESENTATIONS

Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, Ming Zhao. *CloudCache: On-demand Flash Cache Management for Cloud Computing.* Conference on File and Storage Technologies (FAST16).

Swaminathan Sundararaman, Nisha Talagala, Dhananjoy Das, Amar Mudrankit and Dulcardo Arteaga. *Towards Software Defined Persistent Memory.* workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW15).

Jan Lindstrom, Dhananjoy Das, Torben Mathiasen, Dulcardo Arteaga, Nisha Talagala. *NVM Aware MariaDB Database System.* 4th IEEE Non-Volatile Memory System and Applications Symposium (NVMSA15).

Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindstrom. *NVM Compression Hybrid Flash-Aware Application Level Compression.* workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW14).

Dulcardo Arteaga, Ming Zhao. *Client-side Flash Caching for Cloud Systems.* 7th ACM International Systems and Storage Conference (SYSTOR'14).

Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, Seetharami Seelam. *vPFS: Virtualization-based Bandwidth Management for Parallel Storage Systems.* 28th IEEE Conference on Massive Data Storage (MSST12).

Dulcardo Arteaga, Ming Zhao. *Towards Scalable Application Checkpointing with*

Parallel File System Delegation. 6th IEEE International Conference on Networking, Architecture, and Storage (NAS11).

Yonggang Liu, Renato Figueiredo, Dulcardo Arteaga, Yiqi Xu, Ming Zhao. *Towards Simulation of Parallel File System Scheduling Algorithms with PFSsim.* 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI, co-held with MSST11).

Dulcardo Arteaga, Ming Zhao, Chen Liu, Pollawat Thanarungroj, Lichen Weng. *Cooperative Virtual Machine Scheduling on Multi-core Multi-threading Systems A Feasibility Study.* Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Cloud (MASVDC, co-held with MICRO10).

Yiqi Xu, Lixi Wang, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, and Renato Figueiredo. *Virtualization-based Storage Management for High-end Computing Systems.* 5th Petascale Data Storage Workshop (PDSW, co-held with SC10).