Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

7-24-2012

Improving Caches in Consolidated Environments

Ricardo Koller
Florida International University, rkoll001@fiu.edu

Follow this and additional works at: http://digitalcommons.fiu.edu/etd

Recommended Citation
Koller, Ricardo, "Improving Caches in Consolidated Environments" (2012). *FIU Electronic Theses and Dissertations*. 708.
http://digitalcommons.fiu.edu/etd/708

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

IMPROVING CACHES IN CONSOLIDATED ENVIRONMENTS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Ricardo Koller

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Ricardo Koller, and entitled Improving Caches in Consolidated Environments, 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.

Giri Narasimhan

Ming Zhao

Chen Liu

Murali Vilayannur

Raju Rangaswami, Major Professor

Date of Defense: July 24, 2012

The dissertation of Ricardo Koller is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

© Copyright 2012 by Ricardo Koller

All rights reserved.

ACKNOWLEDGMENTS

Muchas gracias a mis padres: Silvia, James, Norma y Ludvik.

Many thanks to my collaborators: Raju Rangaswami, Akshat Verma, Ali Mashtizadeh and Murali Vilayannur. Akshat contributed to formulating the ERSS tree model. Ali and Murali helped in designing and coding the SSD cache for virtual machines. And Prof. Raju who, besides being my advisor and a great friend, contributed to imagining, designing, and evaluating all the systems presented in this thesis.

DEDICATION

Para la princesa y mi gorda.

ABSTRACT OF THE DISSERTATION

IMPROVING CACHES IN CONSOLIDATED ENVIRONMENTS

by

Ricardo Koller

Florida International University, 2012

Miami, Florida

Professor Raju Rangaswami, Major Professor

Memory (cache, DRAM, and disk) is in charge of providing data and instructions to a computer's processor. In order to maximize performance, the speeds of the memory and the processor should be equal. However, using memory that always match the speed of the processor is prohibitively expensive. Computer hardware designers have managed to drastically lower the cost of the system with the use of memory caches by sacrificing some performance. A cache is a small piece of fast memory that stores popular data so it can be accessed faster. Modern computers have evolved into a hierarchy of caches, where a memory level is the cache for a larger and slower memory level immediately below it. Thus, by using caches, manufacturers are able to store terabytes of data at the cost of cheapest memory while achieving speeds close to the speed of the fastest one.

The most important decision about managing a cache is what data to store in it. Failing to make good decisions can lead to performance overheads and over-provisioning. Surprisingly, caches choose data to store based on policies that have not changed in principle for decades. However, computing paradigms have changed radically leading to two noticeably different trends. First, caches are now consolidated across hundreds to even thousands of processes. And second, caching is being employed at new levels of the storage hierarchy due to the availability of high-performance flash-based persistent media. This brings four problems. First,

as the workloads sharing a cache increase, it is more likely that they contain duplicated data. Second, consolidation creates contention for caches, and if not managed carefully, it translates to wasted space and sub-optimal performance. Third, as contented caches are shared by more workloads, administrators need to carefully estimate specific per-workload requirements across the entire memory hierarchy in order to meet per-workload performance goals. And finally, current cache write policies are unable to simultaneously provide performance and consistency guarantees for the new levels of the storage hierarchy.

We addressed these problems by modeling their impact and by proposing solutions for each of them. First, we measured and modeled the amount of duplication at the buffer cache level and contention in real production systems. Second, we created a unified model of workload cache usage under contention to be used by administrators for provisioning, or by process schedulers to decide what processes to run together. Third, we proposed methods for removing cache duplication and to eliminate wasted space because of contention for space. And finally, we proposed a technique to improve the consistency guarantees of write-back caches while preserving their performance benefits.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Memory (CPU caches, DRAM, and disk) has become one of the most important design issues of most microprocessors and operating systems, and according to some authors the only important issue [JNW08, Sit96, McK04]. This is due to the ever increasing gap between processor and memory speeds. If the memory system does not provide timely data access to the processor, then processors are kept waiting, doing nothing. Most research has been done on improving processors speeds and to better tolerate slow memory systems. However, for most systems, memory is still a bottleneck and regardless of the processor speed, if the memory is slower, then the processor optimizations are both futile and expensive (both in terms of monetary and energy cost) [Jac09].

Ideally, computers would have a processor reading all of its data and instructions from a flat memory device that matches the processor speed. One such memory device is Static Random Access Memory (SRAM). Building a computer where all data and instructions are stored on SRAM and fetched directly from it to the processor would solve the performance bottleneck and would be a very simple and elegant solution. However, a computer built this way could cost approximately from hundred thousand to 10 million dollars[1]. Consequently, hardware manufacturers have sacrificed performance to achieve a lower price at a very good trade-off: minimal performance impact for huge price reductions.

There are many types of memory, all of them store information using some special physical property: disks use magnetism and semiconductor memories use electricity. These memories offer specific trade-offs between cost, capacity, and speed of access. For example, disks are cheap, large, and slow; and semiconductor memories

[1] 100 GB at roughly 1 to 100 $ per MB [JNW08]

like SRAM are expensive, small, and fast. Computers have evolved into using a combination of many memory types and are now able to store terabytes of data at the cost of the cheapest memory type while achieving speeds close to the speed of the fastest memory type.

Computer manufacturers are capable of achieving high speed at low price due to one crucial engineering concept: *caches*. A cache is a small piece of fast memory that stores popular data so it can be accessed faster. Caches are designed to take advantage of a general phenomenon seen in applications' data and instructions: temporal locality. This phenomenon states that if data is accessed, it is very likely that it will be accessed again in the near future. Therefore, if a cache stores the data recently and frequently used in the past, it is very likely that cached data will be used again. By doing this, memory systems are able to efficiently use small, fast, and expensive caches for large terabytes of cheap memory storage.

Modern computers use the idea of caching in a hierarchy: there are several levels of memory and a memory level is a cache for a larger and slower memory level immediately below it. We now examine the most commonly used memory hierarchy. The first level is the processor which is supplied data and instructions from memory levels in the following order: CPU caches, main memory, and disk.

The access time of a level compared to the one immediately below is 10 to 1000 times smaller. That means that if the cache is unable to store data that is going to be used next, then the processor has to wait 10 to 1000 times more until it is retrieved from the level below. Sometimes, data can even be across a wide-area-network and accessing it can take several seconds, or data cached could be the result of a very lengthy computation. Therefore the cost of re-fetching the data or perform the slow computation again is too high: there is limited space and the cost of a cache miss is too high. Therefore, special care needs to be taken in choosing which blocks to store

in caches at any time. Surprisingly, caches at all levels choose data to store based on policies that have not changed much in principle for decades. For instance, CPU and buffer caches are still managed using the Least Recently Used (LRU) policy introduced thirty years ago [Car81].

Since the introduction of policies like LRU, computing has changed radically, with the biggest change being that nowadays caches are shared across hundreds or thousands of simultaneous workloads. This change has introduced three problems:

- *Duplicated content in caches.* Having more workloads increase the chances of having blocks of data with the exact same content at the same cache level. This can waste precious cache space at all levels.
- *Unknown cache requirements.* In order to meet the performance goals of applications, users need to provision cache space for each workload sharing the cache with other workloads. For this, a precise estimation of usage across all levels of the memory hierarchy becomes necessary. Existing techniques for cache estimation have severe drawbacks when used for consolidated caches.
- *Contention for shared caches.* We observed experimentally that caches managed with traditional unified cache replacement policies can lead to wasted space, and that this wastage increases linearly with the number of workloads.

The second big change in computing paradigms is that nowadays caching is being employed at new levels of the storage hierarchy due to the availability of high-performance flash-based persistent media. Write-policies are not capable of providing both performance and consistency guarantees.

We believe that these are significant problems and that solving them have the potential of further bridging the gap between processor and memory system speeds.

Additionally, by modeling a workload's behavior and requirements in terms of cache space we can make systems more predictable and workloads truly isolated.

This thesis makes the following contributions.

1. A technique to eliminate duplicated content from caches and use this new space for other data, therefore improving performance.
2. A generalized and unified model for applications' cache requirements across the memory hierarchy. Additionally, we model the effect of sharing a cache for several applications.
3. A technique to reduce wastage in shared caches. Previous research on partitioning only applies to CPU and buffer caches [SRD04, GJMT03]. We propose a generalized partitioning method applicable to any cache in the memory hierarchy. And finally.
4. A technique to improve the consistency guarantees of write-back caches while preserving their performance benefits.

The remaining chapters of this thesis are organized as follows. In chapter 2, we state the thesis contributions and significance. Chapter 3 is a description of the technique to eliminate duplication from caches. In chapter 4, we present the generalized and unified model for cache requirements. Chapter 5 describes the partitioning technique to reduce contention. In chapter 6, we show the design and implementation of the write policies capable of achieving performance and consistency guarantees. And finally, in chapters 7 and 8 we present related and future work.

CHAPTER 2

THESIS OVERVIEW

In this chapter we state the contributions and significance of this thesis.

2.1 Thesis Statement

We propose improving the state-of-the-art in caching by:

1. reducing duplication in data content to improve caching efficiency,
2. modeling applications' cache usage across the memory hierarchy and the effect of contention when they are shared,
3. solving the cache contention problem through cache partitioning and isolation of workloads, and
4. achieving consistency guarantees for file systems, while achieving performance comparable to write-back caches, at the cost of controlled staleness.

2.2 Thesis Contributions

Recent trends in computing have increased the number of workloads sharing caches. One such trends is the use of virtualization by which several virtual machines themselves often running hundreds of processes are consolidated into the same physical machines, and therefore sharing all the CPU caches and the main memory used as cache for disk. In this thesis we model the effects of such consolidation and propose solutions to improve hit ratios of caches, improve performance, and ultimately reduce cost. We propose three specific contributions.

The first contribution is to study the amount of duplicated content present in consolidated buffer caches (RAM caches for disk) and find ways of using it for improving cache hit-rates. We studied duplication of data in the I/O path at the

RAM buffer caches for disk data through the use of production traces. We found that there is a substantial amount of duplicated data being cached, and that same content data is accessed more frequently than the same data location. These two observations then motivated the construction of a *content addressed cache* which provides higher hit-rates than traditional location addressed caches. We present the design and evaluation of this system in Section 3.

The second contribution is the creation of application cache usage models across the entire memory hierarchy and being able to predict the resulting effect on performance when a set of applications contend for caches. We introduce the concept of Effective Reuse Set Size (ERSS) Tree as the basis for the modeling in Section 4. We then study methods for measuring and modeling it across all levels of the memory hierarchy. We specifically develop methods for measuring ERSS for CPU, RAM, and external memory caches. The main focus of study is related to the adverse effect that consolidation has on caches at any level of the memory hierarchy. We analytically prove certain properties of cache replacement policies such as cache wastage being greater or equal than zero regardless of the replacement policy used.

The third contribution addresses the reduction of cache space wastage because of cache sharing. Several researchers have proposed partitioning caches in order to reduce wastage. However, the partitioning methods proposed in the literature make assumptions only applicable to the first levels of the memory hierarchy: CPU and RAM caches. We found that most of these assumptions and related methods have large errors when applied to external memory caches. In chapter 5 we propose a partitioning method for Solid State Drives (SSD) caches (a type of external memory cache). Additionally, in chapter 6 we present two write-policies capable of providing consistency guarantees while still performing similarly to write-back, this at the cost of controlled staleness.

2.3 Thesis Significance

2.3.1 Using data de-duplication to improve caching efficiency

We observed that the amount of duplicated data in shared buffer caches is such that removing it can boost applications' performance by 10% to 4x when compared to conventional location-addressed caches. This is because by removing duplicates and having the cache addressed by content we can make space for other cached data which would otherwise miss the cache.

The impact of this for applications and users is that they can get an increase in performance without augmenting the size of the cache. A nice property of using a content-addressed cache is that miss rates will always be lower or equal than location-addressed caches: the same content can be in more than one location, but not vice versa. And because running many workloads on the same machine is more the norm than an exception, it is very likely that for most systems, same content resides in more than one location. Therefore, we expect performance gains for the majority of systems.

2.3.2 Modeling cache usage

An application requires resources at various memory levels in order to meet its performance objective. Administrators are constantly provisioning resources for applications and need accurate information about their requirements to do this well. One such resource is memory, which translates to memory requirements at all levels of the hierarchy. Not meeting cache requirements can lead to substantial performance degradation. For example, Verma *et al.* have observed a performance impact of up to a factor of 4 due to CPU-cache-unaware allocations for many HPC applications [VAN08b].

Further, memory provisioning needs to deal with the problem of multiple applications contending for shared levels of the memory hierarchy. We observed that this contention can result in a large amount of wasted cache space. We propose including this in the memory models by accurately characterizing the amount of cache resources required by each application including the over-provisioning required to address contention and ensure performance isolation between the applications.

2.3.3 Reducing cache contention

Server consolidation (e.g. via virtualization) is gaining acceptance as the solution to overcome server sprawl by replacing many low utilization servers with a few highly utilized servers. However, contention for the memory system can be a source of unexpected impact to an application's performance due to consolidation. We noticed that the source of this performance impact is some wasted space allocated in the caches that is not usable by any workload. We additionally observed that this wasted space increases linearly with the number of workloads, and that it exists for any cache replacement policy.

This problem was not serious thirty years ago, but in current over-consolidated scenarios where thousands of applications are sharing the same caches, this wasted space can be as big as the cache required by one of the applications contending for cache space. By managing caches more efficiently, we would have space to host one or more additional applications per host thereby reducing costs. Our proposed approach to cache partitioning eliminates wasted space in caches entirely.

2.3.4 Guaranteeing consistency in write-back policies

There are two types of write-policies for caches: write-back and write-through. Write-back is a policy optimized for write accesses, which does not provide any guarantee of consistency. Write-through, on the other hand, does not optimize writes, and therefore performs worse than write-back for some workloads. It does, at least, provide full guarantees of data consistency.

Modern computers use write-back caches for most of the memory hierarchy: CPU caches and RAM [Int09, LSB08]. In contrast, SSD caches typically use write-through caches because they are placed below file systems which already guarantee some form of consistency. The problem with using write-through caches is that we observed that for some transactional workloads, write-back caches allow the applications to achieve 8 times more transactions per second. Our proposed solution achieves performance comparable to write-back with some consistency guarantees by controlling a third dimension of cache behavior: staleness.

CHAPTER 3

CONTENT ADDRESSED CACHING

In this chapter, we present a storage optimization that utilizes content similarity for improving performance by eliminating duplicated content in caches. This technique is motivated by our observations with I/O workload traces obtained from actively-used production storage systems, all of which revealed surprisingly high levels of content similarity for both stored and accessed data.

Duplication of data in primary storage systems is quite common due to the technological trends that have been driving storage capacity consolidation. The elimination of duplicate content at both the file and block levels for improving storage space utilization is an active area of research [CAVL09, JDT05, KDLT04, LEB+09, QD02, RCP08, ZLP08]. Indeed, eliminating most duplicate content is inevitable in capacity-sensitive applications such as archival storage for cost-effectiveness. On the other hand, there exist systems with moderate degree of content similarity in their primary storage such as email servers, virtualized servers, and NAS devices running file and version control servers. In case of email servers, mailing lists, circulated attachments and SPAM can lead to duplication. Virtual machines may run similar software and thus create co-located duplicate content across their virtual disks. Finally, file and version control systems servers of collaborative groups often store copies of the same documents, sources and executables. In such systems, if the degree of content similarity is not overwhelming, eliminating duplicate data may not be a primary concern.

Gray and Shenoy have pointed out that given the technology trends for price-capacity and price-performance of memory/disk sizes and disk accesses respectively, disk data must "cool" at the rate of 10X per decade [GS00]. They suggest data replication as a means to this end. An instantiation of this suggestion is *intrinsic*

replication of data created due to consolidation as seen now in many storage systems, including the ones illustrated earlier. Here, we refer to intrinsic (or application/user generated) data replication as opposed to forced (system generated) redundancy such as in a RAID-1 storage system. In such systems, capacity constraints are invariably secondary to I/O performance.

We analyzed on-disk duplication of content and I/O traces obtained from three varied production systems at FIU that included a virtualized host running two department web-servers, the department email server, and a file server for our research group. We made three observations from the analysis of these traces. First, our analysis revealed significant levels of duplicate content in the storage medium and in the portion of the medium that is accessed by the I/O workload. We define these similarity measures formally in § 3.1. Second, we discovered a consistent and marked discrepancy between *reuse distances* [MGST70a] for sector and content in the I/O accesses on these systems indicating that content is reused more frequently than sectors. Third, there is significant overlap in content accessed over successive intervals of longer time-frames such as days or weeks.

Based on these observations, we explore the premise that intrinsic content similarity in storage systems and access to replicated content within I/O workloads can both be utilized to improve I/O performance. In doing so, we design and evaluate a storage optimization that utilizes content similarity to eliminate I/O operations altogether. The main mechanism is *content-addressed caching*, which uses the popularity of "data content" rather than "data location" of I/O accesses in making caching decisions.

We evaluated a Linux implementation of a content-addressed cache for workloads from the three systems described earlier. Performance improvements measured as the reduction in total disk busy time in the range 28-47% were observed across these

workloads. Content-addressed caching increased memory caching effectiveness by at least 10% and by as much as 4X in cache hit rate for read operations.

We also measured the memory and CPU overheads and found these to be nominal.

In Section 3.1, we make the case for I/O deduplication. We elaborate on a specific design and implementation of its techniques in Section 3.2. We perform a detailed evaluation of improvements and overhead for three different workloads in Section 3.3. We discuss related research in Section 3.4, and finally conclude with directions for future work.

3.1 Motivation

In this section, we investigate the nature of content similarity and access to duplicate content using workloads from three production systems that are in active, daily use at the FIU Computer Science department. We collected I/O traces downstream of an active page cache from each system for a duration of three weeks. These systems have different I/O workloads that consist of a virtual machine running two web-servers (*web-vm* workload), an email server (*mail* workload), and a file server (*homes* workload). The *web-vm* workload is collected from a virtualized system that hosts two CS department web-servers, one hosting the department's online course management system and the other hosting the department's web-based email access portal; the local virtual disks which were traced only hosted root partitions containing the OS distribution, while the http data for these web-servers reside on a network-attached storage. The *mail* workload serves user INBOXes for the entire Computer Science department at FIU. Finally, the *homes* workload is that of a NFS server that serves the home directories of our small-sized research group; activities represent those of a typical researcher consisting of software development, testing,

and experimentation, the use of graph-plotting software, and technical document preparation.

Workload type	File Sys. size [GB]	Unique reads [GB]			Unique writes [GB]			File System accessed
		Total	Sectors	Content	Total	Sectors	Content	
web-vm	70	3.40	1.27	1.09	11.46	0.86	4.85	2.8%
mail	500	62.00	29.24	28.82	482.10	4.18	34.02	6.27%
homes	470	5.79	2.40	1.99	148.86	4.33	33.68	1.44%

Table 3.1: Summary statistics of one week I/O workload traces obtained from three different systems.

Key statistics related to these workloads are summarized in Table 3.1. The mail server is a heavily used system and generates a highly-intensive I/O workload in comparison to the other two. However, some uniform trends can be observed across these workloads. A fairly small percentage of the total file system data is accessed during the entire week (1.44-6.27% across the workloads), representing small working sets. Further, these are write-intensive workloads. While it is therefore important to optimize write I/O operations, we also note that most writes are committed to persistent storage in the background and do not affect user-perceived performance directly. Optimizing read operations, on the other hand, has a direct impact on user-perceived performance and system throughput because this reduces the waiting time for blocked foreground I/O operations. For read I/O's, we observe that in each workload, the unique content accessed is lesser than the unique locations that are accessed on the storage device. Notice that these are the unique number of content and sectors reads, not the total number of accesses. This is why the same content is accessed more than once and therefore unique sector reads are equal or higher than unique content reads. These observation directly motivates the three techniques of our approach as we elaborate next.

The systems of interest in our work are those in which there are patterns of work shared across more than one mechanism within a single system. A *mechanism*

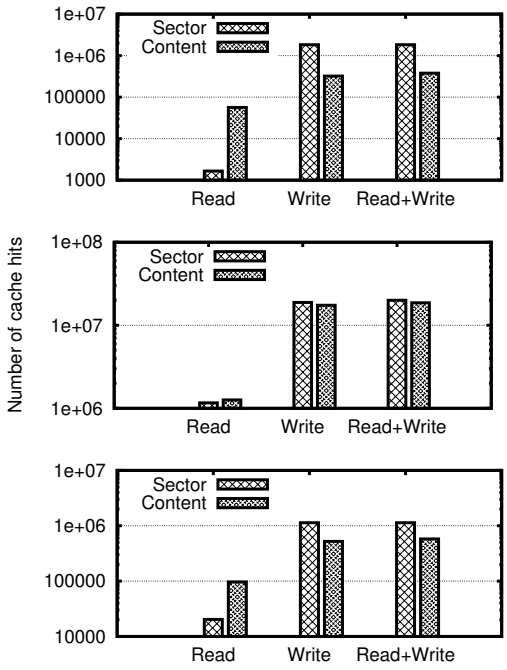

Figure 3.1: Page cache hits for the web-vm (top), mail (middle), and homes (bottom) workloads. A single day trace was used with an infinite cache assumption.

represents any active entity, such as a single thread or process or an entire virtual machine. Such duplicated mechanisms also lead to intrinsic duplication in content accessed within the respective mechanisms' I/O operations. Duplicate content, however, may be independently managed by each mechanism and stored in distinct locations on a persistent store. In such systems, traditional storage-location (sector) addressed caching can lead to content duplication in the cache, thus reducing the effectiveness of the cache.

Figure 3.1 shows that cache hit ratio (for read requests) can be improved substantially by using a content-addressed cache instead of a sector-addressed one. Notice that this is a count of accesses to the same content or same sector, and not the total number of accesses. While write I/Os leading to content hits could be eliminated for improved performance, we do not explore them in this thesis. A greater number of sector hits with write I/Os are due to journaling writes by the file system,

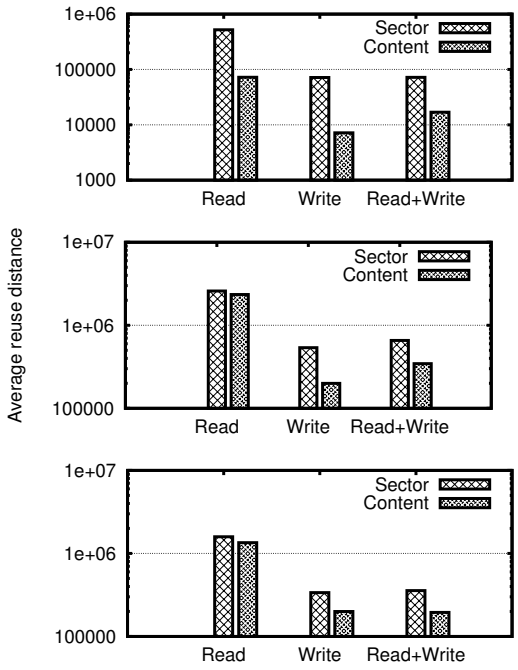

Figure 3.2: Contrasting content and sector reuse distances for the web-vm (top), mail (middle), and homes (bottom) workloads.

repeatedly overwriting locations within a circular journal space.

For further analysis, we define the *average sector reuse distance* for a workload as the average number of requests between successive requests to the same sector. The *average content reuse distance* is defined similarly over accesses to the same content. Figure 3.2 shows that the average reuse distance for content is smaller than for sector for each of the three workloads that we studied for both read and write requests. For such workloads, data addressed by content can be cache-resident for lesser time yet be more effective for servicing read requests than if the same cached data is addressed by location. Write requests on the other hand do not depend on cache hits since data is flushed to rather than requested from the storage system. These observations and those from Figure 3.1 motivate *content-addressed caching*.

3.2 Design

In this section, we start with an overview of the system architecture and then present the various design choices and rationale behind constructing the content-address cache.

3.2.1 Architectural Overview

An optimization based on content similarity can be built at various layers of the storage stack, with varying degrees of access and control over storage devices and the I/O workload. Prior research has argued for building storage optimizations in the block layer of the storage stack [GUB+08]. We choose the block layer for several reasons. First, the block interface is a generic abstraction that is available in a variety of environments including operating system block device implementations, software RAID drivers, hardware RAID controllers, SAN (e.g., iSCSI) storage devices, and the increasingly popular storage virtualization solutions (e.g., IBM SVC [IBM], EMC Invista [EMC], NetApp V-Series [Net]). Consequently, optimizations based on the block abstraction can potentially be ported and deployed across these varied platforms. In the rest of the chapter, we develop an operating system block device oriented design and implementation.. Second, the simple semantics of block layer interface allows easy I/O interception, manipulation, and redirection. Third, by operating at the block layer, the optimization becomes independent of the file system implementation, and can support multiple instances and types of file systems. Fourth, this layer enables simplified control over system devices at the block device abstraction, allowing an elegantly simple implementation of selective duplication that we describe later. Finally, additional I/Os generated can leverage

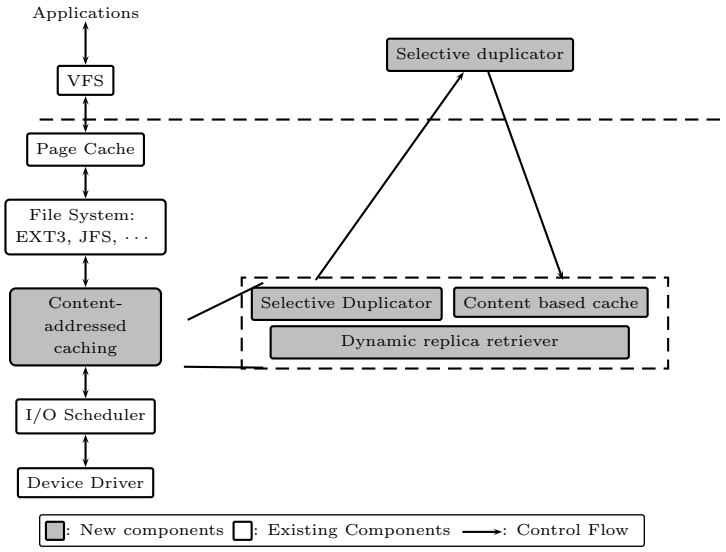

Figure 3.3: System Architecture.

I/O scheduling services, thereby automatically addressing the complexities of block request merging and reordering.

Figure 3.3 presents the architecture of content-addressed caching for a block device in relation to the storage stack within an operating system. We augment the storage stack's block layer with additional functionality, which we term the *I/O dedup layer*, to implement the three major mechanisms: the content-addressed cache, the dynamic replica retriever, and the selective duplicator. The *content-addressed cache* is the first mechanism encountered by the I/O workload, which filters the I/O stream based on hits in a content-addressed cache. The *dynamic replica retriever* subsequently optionally redirects the unfiltered read I/O requests to alternate locations on the disk to avail the best access latencies to requests. The *selective duplicator* is composed of a *kernel* sub-component that tracks content accesses to create a candidate list of content for replication, and a *user-space* process that runs during periods of low disk activity and populates replica content in scratch space distributed across the entire disk. Thus, while the kernel components run

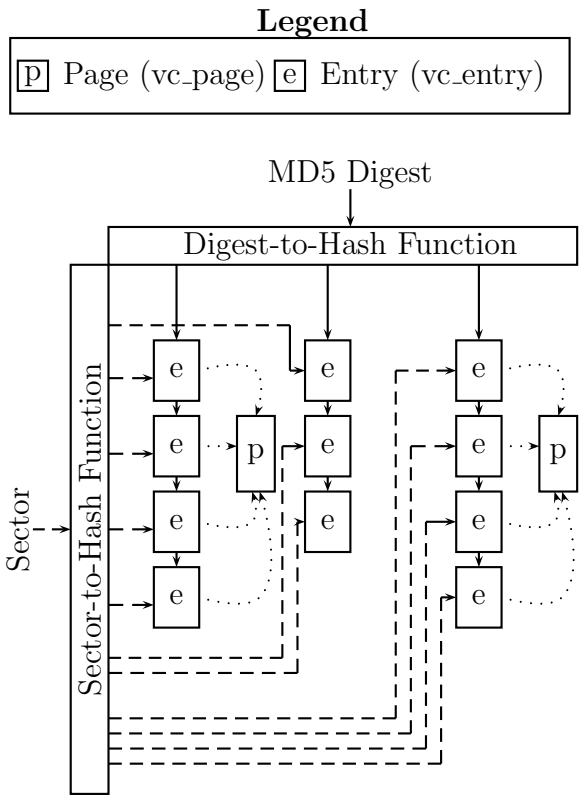

Figure 3.4: Data structure for the content-addressed cache. The cache is addressable by both sector and content-hash. vc_entries are unique per sector. Solid lines between vc_entries indicate that they may have the same content (they may not in case of hash function collisions.) Dotted lines form a link between a sector (vc_entry) and a given page (vc_page.) Note that some vc_entries do not point to any page – there is no cached content for these entries. However, this indicates that the linked vc_entries have the same data on disk. This happens when some of the pages are evicted from the cache. Additionally, pages form an LRU list.

continuously, the user-space component runs sporadically. Separating out the actual replication process into a user-level thread allows greater user/administrator control over the timing and resource consumption of the replication process, an I/O resource-intensive operation. Next, we elaborate on the design of the content addressed caching mechanism.

3.2.2 Content addressed caching

Building a content-addressed cache at the block layer creates an additional buffer cache separate from the virtual file system (VFS) cache. Requests to the VFS cache are sector-based while those to the content-addressed cache cache are both sector- and content-addressed. The content-addressed cache layer only sees the read requests for sector misses in the VFS cache. We discuss exclusivity across these caches shortly. In the content-addressed cache layer, read requests identified by sector locations are queried against a dual sector- and content-addressed cache for hits before entering the I/O scheduler queue or being merged with an existing request by the I/O scheduler. Population of the content-addressed cache occurs along both the read and write paths. In case of a cache miss during a read operation, the I/O completion handler for the read request is intercepted and modified to additionally insert the data read into the content-addressed cache after I/O completion only if it is not already present in the cache and is important enough in the LRU list to be cached. A write request to a sector which had contained duplicate data is simply removed from the corresponding duplicate sector list to ensure data consistency for future accesses. The new data contained within write requests is optionally inserted into the content-addressed cache (if it is sufficiently important) in the onward path before entering the request into the I/O scheduler queue to keep the content cache up-to-date with important data.

The in-memory data structure implementing the content-addressed cache supports look-up based on both sector and content-hash to address read and write requests respectively. Entries indexed by content-hash values contain a sector list (list of sectors in which the content is replicated) and the corresponding data if it was entered into the cache and not replaced. Cache replacement only replaces the

content field and retains the sector-list in the in-memory content-cache data structure. For read requests, a sector-based lookup is first performed to determine if there is a cache hit. For write requests, a content-hash based look-up is performed to determine a hit and the sector information from the write request is added to the sector-list. Figure 3.4 describes the data structure used to manage the content-addressed cache. A write to a sector that is present in a sector-list indexed by content-hash is simply removed from the sector list and inserted into a new list based on the sector's new content hash. It is important to also point out that our design uses a write-through cache to preserve the semantics of the block layer. Ideally, writes hits should dirty the page to be later flushed out to disk by a background process (i.e. pdflush for Linux). However, this current design uses a write-through cache. The main reason is that by having this type of cache, the semantics of writes below the block layer do not change: writes are not postponed. Additionally, the content addressed-cache can be dynamically removed at any time without any delay or inconsistency. Next, we discuss some practical considerations for our design.

Since the content cache is a second-level cache placed below the file system page cache or, in case of a virtualized environment, within the virtualization mechanism, typically observed recency patterns in first level caches are lost at this caching layer. An appropriate replacement algorithm for this cache level is therefore one that captures frequency as well. We propose using Adaptive Replacement Cache (ARC) [MM04] or CLOCK-Pro [JCZ05] as good candidates for a second-level content-addressed cache and evaluate our system with ARC and LRU for contrast.

Another concern is that there can be a substantial amount of duplicated content across the cache levels. There are two ways to address this. Ideally, the content-addressed cache should be integrated into a higher level cache (e.g., VFS page cache) implementation, if possible. However, this might not be feasible in virtualized en-

vironments where page caches are managed independently within individual virtual machines. In such cases, techniques that help make in-memory cache content across cache levels exclusive such as cache hints [LAS+05], demotions [WW02a], and promotions [Gil08] may be used. An alternate approach is to employ memory deduplication techniques such as those proposed in the VMware ESX server [Wal02a], Difference Engine [GLV+08], and Satori [MMHF09]. In these solutions, duplicate pages within and across virtual machines are made to point to the same machine frame with use of an extra level of indirection, such as the shadow page tables. In memory, duplicate content across multiple levels of caches is indeed an orthogonal problem, and any of the referenced techniques could be used as a solution directly within content-addressed caching.

3.2.3 Persistence of metadata

A final issue is the persistence of the in-memory data structure so that the system can retain intelligence about content similarity across system restart operations. Persistence is important for retaining the locations of on-disk intrinsic and artificially created duplicate content so that this information can be restored and used immediately upon a system restart event. We note that while persistence is useful to retain intelligence that is acquired over a period of time, "continuous persistence" of metadata in a content-addressed cache is not necessary to guarantee the reliability of the system, unlike other systems such as the eager writing disk array [ZYKW02] or doubly distorted mirroring [OS93]. In this sense, *selective duplication* is similar to the opportunistic replication as performed by FS2 [HHS05] because it tracks updates to replicated data in memory and only guarantees that the primary copy of data blocks are up-to-date at any time. While persistence of the in-memory data is not implemented in our prototype yet, guaranteeing such persistence is relatively

straightforward. Before the content-addressed cache kernel module is unloaded (occurring at the same time the managed file system is unmounted), all in-memory data structure entries can be written to a reserved location of the managed scratch-space. These can then be read back to populate the in-memory metadata upon a system restart operation when the kernel module is loaded into the operating system.

3.3 Experimental Evaluation

In this section, we evaluate the performance impact of using a content-addressed cache. We also evaluate the CPU and memory overhead incurred by an content-addressed cache. We used the block level traces for the three systems that were described in detail in § 3.1 for our evaluation. The traces were replayed as block traces in a similar way as done by blktrace [Axb07]. Blktrace could not be used as-is since it does not record content information; we used a custom Linux kernel module to record content-hashes for each block read/written in addition to other attributes of each I/O request. Additionally, the blktrace tool btreplay was modified to include traces in our format and replay them using provided content. Replay was performed at a maximum acceleration of 100x, with care being taken in each case to ensure that block access patterns were not modified as a result of the speedup. Measurements for actual disk I/O times were obtained with per-request block-level I/O tracing using blktrace and the results reported by it. Finally, all trace playback experiments were performed on a single Intel(R) Pentium(R) 4 CPU 2.00GHz machine with 1 GB of memory and a Western Digital disk WD5000AAKB-00YSA0 running Ubuntu Linux 8.04 with kernel 2.6.20.

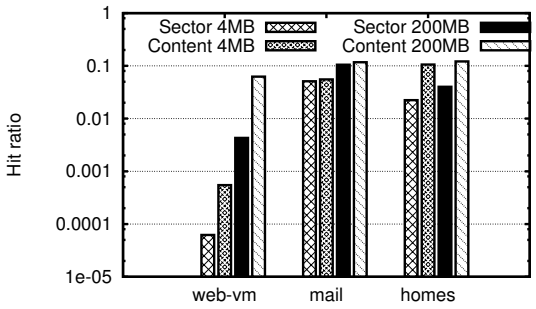

Figure 3.5: Per-day page cache hit ratio for content- and sector- addressed caches for read operations. The total number of pages read are 0.18, 2.3, and 0.23 million respectively for the web-vm, mail and homes workloads. The numbers in the legend next to each type of addressing represent the cache size.

3.3.1 Evaluating performance

In our first experiment, we evaluated the effectiveness of a content-addressed cache against a sector-addressed one. The primary difference in implementation between the two is that for the sector-addressed cache, the same content for two distinct sectors will be stored twice. We fixed the cache size in both variants to one of two different sizes, 1000 pages (4MB) and 50000 pages (200MB). We replayed two weeks of the traces for each of the three workloads; the first week warmed up the cache and measurements were taken during the second week. Figure 3.5 shows the average per-day cache hit counts for read I/O operations during the second week when using an *adaptive replacement cache* (ARC) in two modes, content and sector addressed.

This experiment shows that there is a large increase in per-day cache hit counts for the web and the home workloads when a content-addressed cache is used (relative to a sector-addressed cache). The first observation is that improvement trends are consistent across the two cache sizes. Both cache implementations benefit substantially from a larger cache size except for the *mail* workload, indicating that *mail* is not a cache-friendly workload validated by its substantially larger working set and workload I/O intensity (as observed in Section 3.1). The *web-vm* workload

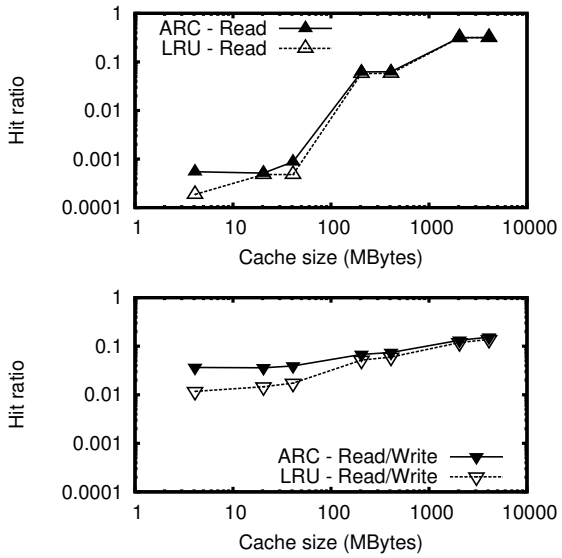

Figure 3.6: Comparison of ARC and LRU content-addressed caches for pages read only (top) and pages read/write operations (bottom). A single day trace (0.18 million page reads and 2.09 million page read/writes) of the web workload was used as the workload.

shows the biggest increase with an almost 10X increase in cache hits with a cache of 200MB compared to the home workload which has an increase of 4X. The *mail* workload has the least improvement of approximately 10%.

We performed additional experiments to compare an LRU implementation with the ARC cache implementation (used in the previous experiments) using a single day trace of the *web-vm* workload. Figure 3.6 provides a performance comparison of both replacement algorithms when used for a content-addressed cache. For small and large cache sizes, we observe that ARC is either as good or more effective than LRU with ARC's improvement over LRU increasing substantially for write operations at small to moderate cache sizes. More generally, this experiment suggests that the performance improvements for a content-addressed cache are sensitive to the cache replacement mechanism, which should be chosen with care.

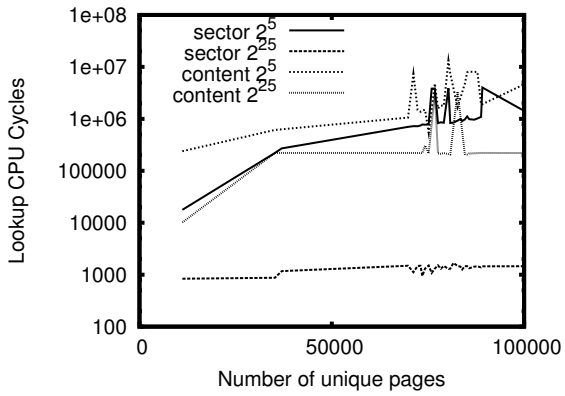

Figure 3.7: Overhead of content and sector lookup operations with increasing size of the content-addressed cache.

3.3.2 Evaluating Overhead

While the gains due to addressing a cache by content are promising, it incurs resource overhead. Specifically, the implementation uses content- and sector- addressed hash-tables to simplify lookup and insert operations into the content-addressed cache. We evaluate the CPU overhead for insert/lookup operations and memory overhead required for managing hash-table metadata.

CPU Overhead

To evaluate the overhead, we measured the average number of CPU cycles required for lookup/insert operations as we vary the number of unique pages (i.e., size) in the content-addressed cache (i.e., cache size) for a day of the *web* workload. Figure 3.8 depicts these overheads for two cache configurations, one configured with 2^{25} buckets in the hash tables and the other with 2^5 buckets. Read operations perform a sector lookup and additionally content lookup in case of a miss for insertion. Write operations always perform a sector and content lookup due to our write-through cache design. Content lookups need to first compute the hash for the page contents which takes around $100K$ CPU cycles for MD5. With few buckets (2^5) lookup times approach O(N) where N is the size of the hash-table. However,

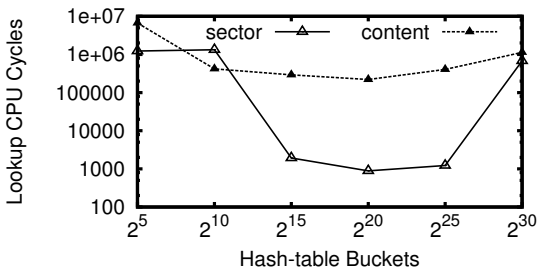

Figure 3.8: Overhead of sector and content lookup operations with increasing hash-table bucket entries.

given enough hash-table buckets (2^{25}), lookup times are O(1).

Next, we examined the sensitivity to the hash-table bucket entries. As the number of buckets are increased, the lookup times decrease as expected due to reduction in collisions, but beyond 2^{20} buckets, there is an increase. We attribute this to L2 cache and TLB misses due to memory fragmentation, under-scoring that hash-table bucket sizes should be configured with care. In the sweet spot of bucket entries, the lookup overhead for both sector and content reduces to 1K CPU cycles or less than 1μs for our 2GHz machine. Note that the content lookup operation includes a hash computation, which inflates its cycles requirement by at least $100K$.

Memory Overhead

The management of a content-addressed cache introduces memory overhead for managing metadata for the content-addressed cache. Specifically, the memory overhead is dictated by the size of the cache measured in pages (P), the degree of *workload static similarity* (WSS), and the configured number of buckets in the hash tables (HTB), which also determine the lookup time, as we saw earlier. WSS is defined as the average number of copies per block accessed by the I/O workload. In our current unoptimized implementation, the memory overhead in bytes (assuming 4 byte pointers and 4096 byte pages) :

$$mem(P, WSS, HTB) \quad = 13 * P + 36 * P * WSS + 8 * HTB$$

These overheads include 13 bytes per-page to store the metadata for a a specific page content (*vc_page*), 36 bytes per page per duplicated entry (*vc_entry*), and 8 bytes per hash-table entry for the corresponding linked list. For a 1GB content cache (256K pages), a static similarity of 4, and a hash-table of size 1 million entries, the metadata overhead is ∼48MB or approximately 4.6%.

3.4 Related work

Content similarity in both memory and archival storage have been investigated in the literature. Memory deduplication has been explored before in the VMware ESX server [Wal02a], Difference Engine [GLV+08], and Satori [MMHF09], each aiming to eliminate duplicate in-memory content both within and across virtual machines sharing a physical host. Of these, Satori has apparent similarities to our work because it identifies candidates for in-memory deduplication as data is read from storage. Satori runs in two modes: content-based sharing and copy-on-write disk sharing. For content-based sharing, Satori uses content-hashes to track page contents in memory read from disk. Since its goal is not I/O performance optimization, it does not track duplicate sectors on disk and therefore does not eliminate duplicated I/Os that would read the same content from multiple locations. In copy-on-write disk sharing, the disk is already configured to be copy-on-write enabling the sharing of multiple VM disk images on storage. In this mode, duplicated I/Os due to multiple VMs retrieving the same sectors on the shared physical disk would be eliminated in the same way as a regular sector-addressed cache would do. In contrast, our work eliminates duplicated I/Os by retrieving their content irrespective of where they

reside on storage. Additionally, our work improves I/O performance by reducing head movement. Thus, the contributions of Satori are complementary to our work and can be used simultaneously.

Data deduplication in archival storage has also gained importance in both the research and industry communities. Current research on data deduplication uses several techniques to optimize the I/O overheads incurred due to data duplication. Venti [QD02] proposed by Quinlan and Dorward was the first to propose the use of a content-addressed storage for performing data deduplication in an archival system. The authors suggested the use of an in-memory content-addressed index of data to speed up lookups for duplicate content. Similar content-addressed caches were used in data backup solutions such as Peabody [MIG03] and Foundation [RCP08]. Content-addressed caching is inspired by these works. Recent work by Zhu and his colleagues [ZLP08] suggests new approaches to alleviate the disk bottleneck via the use of Bloom filters [Blo70] and by further accounting for locality in the content stream. The Foundation work suggests additional optimizations using batched retrieval and flushing of index entries and a log-based approach to writing data and index entries to utilize temporal locality [RCP08]. The work on sparse indexing [LEB+09] suggests improvements to Zhu *et al.*'s general approach by exploiting locality in the chunk index lookup operations to further mitigate the disk I/O bottleneck. Content-addressed caching addresses an orthogonal problem, that of improving I/O performance for foreground I/O workload based on the use of duplicates, rather than their elimination. Nevertheless, the above approaches do suggest interesting techniques to optimize the management of a content-addressed index and cache in main-memory that is complementary to and can be used directly within content-addressed caching.

3.5 Summary

System and storage consolidation trends are driving increased duplication of data within storage systems. Past efforts have been primarily directed towards the elimination of such duplication for improving storage capacity utilization. With content-addressed caching, we take a contrary view that intrinsic duplication in a class of systems which are not capacity-bound can be effectively utilized to improve I/O performance – the traditional Achilles' heel for storage systems. The content-addressed caching mechanism increased memory caching effectiveness by increasing cache hit rates by 10% to 4x for read operations when compared to traditional sector-addressed caching.

The work presented in this chapter was reported in ACM Transactions in Storage 2010 [KR10].

As said before, consolidation brings two big problems: data duplication and cache contention. In this chapter we observed that the first problem, can be used as an opportunity to improve performance. In the next chapter, we examine and address the problem of cache contention due to consolidation which introduces a large performance overhead which we could only model and minimize.

CHAPTER 4

MODELING CACHE REQUIREMENTS AND CONTENTION

In order to meet the performance goals of applications, we need to provision cache space across the memory hierarchy. Existing techniques for cache estimation have severe drawbacks when used for consolidated environments. In this chapter, we present a unifying Generalized ERSS Tree Model that characterizes the resource usage at all levels of the memory hierarchy and accurately models the impact of sharing a cache with many workloads.

The ever-increasing gap between processing speed and disk bandwidth ensures that the allocation of resources at all levels of the memory hierarchy (henceforth also referred to simply as *memory levels*) significantly influence the performance of applications. The memory levels used by an application include on-chip caches (L1 and L2), off-chip caches (e.g., L3), DRAM, and other external memory caches (e.g., a flash-based disk cache [krm08]). Characterizing the resource utilization of an application at various memory levels has been an active research area over the years. While the consolidation of multiple applications on a shared hardware is not new, the rise of virtualization has made such systems more the norm than the exception. In virtualized systems, multiple applications run on the same physical server and compete for resources at all memory levels. For instance, it has been shown that contention in a shared cache can lead to performance degradation — as much as 47% [LLD+08] and 75% [VAN08b] depending on the workload. Contrast this problem with duplication (previous chapter's focus). Duplication can be used to boost I/O performance; contention on the other hand is more a problem than an opportunity.

An accurate characterization of the implications of this problem is a prerequisite to ensuring that all applications meet their performance goals [LLD+08, VAN08b,

KRDZ10, VAN08a]. Specifically, there is a need for an accurate characterization of the resource requirement of resident applications within each virtual machine (VM), while carefully taking into account the impact of resource contention due to other applications.

Research in memory characterization can be broadly classified into models of memory usage and techniques that instantiate these models. The modeling work can be summarized using three related but disparate concepts. One of the most popular ways to characterize the main memory usage of an application is the classical working set model [Den80]. The core of this model is the concept of a resident set that defines the set of memory-resident pages of an application at any given point in time. An alternative modeling approach uses the Miss Rate Curve (MRC) [RSG93] of an application to model the influence of the allocated memory cache size on the performance-influencing cache miss rate of the application. MRCs offer an advantage over the working set of being able to model the performance impact of arbitrarily sized caches. Finally, the concepts of phases and phase transitions have been proposed to model the memory resource usage behavior of an application as it changes over time [BM76]. A phase denotes stability of the active data set used by the application and phase transitions indicate a change in application behavior resulting in a change in the set of active data used. and several techniques have been proposed to identify phase transitions [BDB00, DS02, BABD00].

In this work, motivated by the multi-tenancy of applications within virtualized enterprise data centers and clouds, we investigate the problem of characterizing the resource requirement of an application at all levels of the memory hierarchy for accurately provisioning resources in a shared environment. We identify the following properties for a memory model to be relevant in such shared environments.

1. **Address all memory levels**: The degree of sharing in a consolidated environment determines the amount of resources available to an application at each memory level. As opposed to a dedicated system, the resources available to an application at various memory levels in a shared environment do not have clearly demarcated values (e.g, 4KB L1, 2MB L2, 16MB L3 cache). A holistic view of resource consumption across all memory levels during an application's entire lifetime is thus necessary to provision memory resources for the application.

2. **Identify dominant phases**: The phases that are long-running have the greatest impact on application performance. On the other hand, reserving memory resources for short-lived phases, even though they may constitute larger working sets, may not be required. Therefore, model should adequately inform about the lifetime of the phases.

3. **Address impact of contention**: In spite of the copious work on partitioning caches at the hardware or the OS kernel level, solutions for cache partitioning are available only in high-end systems and that too only for certain memory levels. Today's commodity systems do not support flexible cache partitioning at any level. Hence, the model should address the impact of resource contention in a shared environment as a first class concern.

4. **A commodity solution**: Typical virtualized data centers run on commodity systems wherein administrators have little or no control on the internal operations of the guest VMs. Hence, a practically viable model should be built primarily from high level system parameters that do not require intrusive changes to the systems under consideration.

4.1 Gaps in Existing Models and Characterization

Existing memory models are insufficient to adequately model memory usage in a shared environment. First, existing techniques do not adequately capture the distinct phases of memory resource usage during an application's lifetime; working sets typically model the resource requirement for a fixed phase in the application while MRCs unify multiple phases into one, losing important distinctions between their individual resource consumption characteristics. Second, existing models focus on a specific level of memory; working sets are used for main memory [Den80] and MRCs for caches [DS02]. With the increasingly diverse levels of memory resources (L1, L2, L3, main memory, flash, and disks), a unified view of all memory resources is critically important for memory provisioning. For instance, if the working-set of a phase cannot be accommodated in L2, it may be possible to provision for it in L3, and provisioning a greater amount of L3 cache can reduce memory bandwidth requirement. Third, these models were not designed to model the impact of contention between applications which is important for ensuring performance isolation. An application's actual memory requirement may be very small, i.e., *capacity misses* may be close to 0, but it may have a large number of *compulsory misses* (e.g., streaming data access) which would effectively pollute the cache and thus impact other co-located applications significantly. Finally, most existing techniques for identifying phases or to infer the working set or the MRC are fairly intrusive, requiring direct access to the operating system page tables or fragment cache [BDB00] that are only available within the kernel. Typical data centers use commodity software and administrators do not have kernel level access to individual virtual machine instances.

We present the Generalized ERSS Tree Model, a new model for memory usage which both refines and unifies existing models. The Generalized ERSS Tree Model

is based on a novel characterization that we term *Effective Reuse Set Size* (ERSS), which refines the working set by accounting for the reuse of data and architectural advances like prefetching. We define ERSS relative to the miss rate allowing it to model MRC concepts. The ERSS tree additionally captures hierarchical phases and their transitions. Additionally, we introduce two new parameters that intuitively model resource contention. The *Reuse Rate* captures the average rate at which an application reuses its *Effective Reuse Set* and the *Flood Rate* captures the rate at which an application floods a resource with non reusable data. We show that the former captures the vulnerability of an application's performance to competition from other applications, whereas the latter captures the adverse impact of an application on the performance of co-located applications. We overcome significant technical challenges to design a practical methodology for instantiating the model on commodity hardware without access to the target application or operating system. Our methodology uses (i) existing as well as new memory resource partitioning techniques to get the hit/miss rates for applications and (ii) a new phase detection technique that can be built solely from hit and miss rates for all levels of the memory hierarchy. Finally, we demonstrate the use of the model to characterize the amount of memory required to ensure performance isolation for applications in a consolidated environment.

4.2 The Generalized ERSS Tree Model

We now present the Generalized ERSS Tree Model that unifies and refines the classical concepts of working set, miss rate curves, and phases. In line with previous work [BM76], we define a phase as a maximal interval during which a given set of memory lines, each referenced at least once, remain on top of an LRU stack. The model is based on a characterization of the core parameters that determine the

memory capacity requirement of an application within a single phase of its execution. We extend this core concept to a tree-based structural model that characterizes the memory requirements of an application across all of its phases. We finally enrich this model with parameters based on access rates at various memory levels to model resource contention in shared environments. For the rest of the chapter, we use the term *memory line* to denote the basic unit of access, thus denoting both cache lines and memory pages based on the context.

4.2.1 Capacity Related Parameters

A key characteristic of an application's memory usage is the number of memory lines the application requires to avoid capacity misses within a single phase of its execution. This metric assumes that there is no contention from any other application for the resource.

Parameter	Phase i	Duration	InUse Resident Set	Reuse Set	Effective Reuse Set Size	Miss/Hit Ratio	Reuse Rate	Flood Rate
Notation	P_i	θ	IS	RS	$ERSS$	Δ_M	ρ_R	ρ_F

Table 4.1: Phase centric parameters of the model.

Definition 1 In Use Resident Set (IS): The In Use Resident Set for a phase is the set of all distinct virtual memory lines that are accessed during the phase. This notion is the same as the classical working set [Den80], but restricted to a single phase of execution.

While the In Use Resident Set describes the virtual memory lines in use by an application during a phase, we are interested in the physical memory lines that need to be provisioned. A virtual memory line that is not reused may not need a physical memory line to be provisioned for the entire phase which can then be used to host

other lines. Hence, the In Use Resident Set does not adequately capture the amount of physical memory needed by the application.

Definition 2 Reuse Set (RS): The Reuse Set is the subset of In Use Resident Set that is accessed more than once during a phase.

While the *Reuse Set* is a better approximation for the memory requirement of an application, it lacks certain key properties. First, the *Reuse Set* does not capture any capacity that is required to prevent the non-reusable data from evicting the reusable data. The actual memory requirement of the application during a phase may thus be larger than the *Reuse Set*. Second, the *Reuse Set* may contain some virtual memory lines that are reused very infrequently. The performance degradation of the application by not provisioning physical memory for these lines may be negligible and thus these lines do not contribute in a significant way to the effective memory requirement of the application during the phase. Finally, prefetching may hide latencies for accesses that are part of a stream of accesses if the rate of data access is low. This may further reduce the effective number of memory lines required by an application below the *Reuse Set* size. Thus, we define a metric that more accurately captures the amount of memory required by an application within a phase.

Definition 3 Effective Reuse Set Size (ERSS): The Effective Reuse Set Size is the minimum number of physical memory lines that need to be assigned to the program during a phase for it to not exhibit thrashing behavior [Den68a]. The ERSS is defined with respect to a miss/hit ratio Δ_M and is denoted by $ERSS(\Delta_M)$.

Definition 4 Miss/Hit Ratio (Δ_M): This parameter defines the ratio between the number of misses and the number of hits during a phase for a given memory resource allocation.

```
for (idx = 0; idx <A.length ; idx++) {
sum = sum + A[idx];
}
```

Figure 4.1: An illustration of the model concepts.

The new phase centric metrics introduced above are summarized in Table. 4.1. We illustrate the above concepts with an example in Figure 4.1. Consider a phase where a program computes the sum of all the elements in an array in the variable *sum* (Figure 4.1(a)). The Inuse Resident Set (IS) of the program in this phase is the array $\boldsymbol{A}$ and the variables *idx* and *sum*. Since all accesses to $\boldsymbol{A}$ are compulsory misses and only the two variables *idx* and *sum* get reused, the size of the reuse set is 2. The minimum Miss/Hit Ratio (Δ_M) is 1/2 which can be achieved by provisioning physical memory for 3 integers — 2 integers to hold the reuse set and 1 more as a buffer for the current array element. Hence, the $ERSS(1/2)$ is 3 and different from the size of both IS or RS.

Phase transitions in typical programs are abrupt, i.e, the miss/hit ratio is constant for a large range of memory size allocations and increases/decreases significantly with small change in memory resource at a few memory resource allocation sizes [RSG93]. We validate this observation for applications in the NAS benchmark suite [NAS] in Figure 4.2 which reveals sharp knees at a few $ERSS$ values. Since the $ERSS$ for different values of Δ_M around the phase transition (or knee) are similar, one can represent the $ERSS$ for each phase by a single $ERSS$ value corresponding to a default Δ_M; in this work, we used the minimum $ERSS$ at which the derivative of Δ_M w.r.t. memory resource size is less than 0.1 as the default value for that phase.

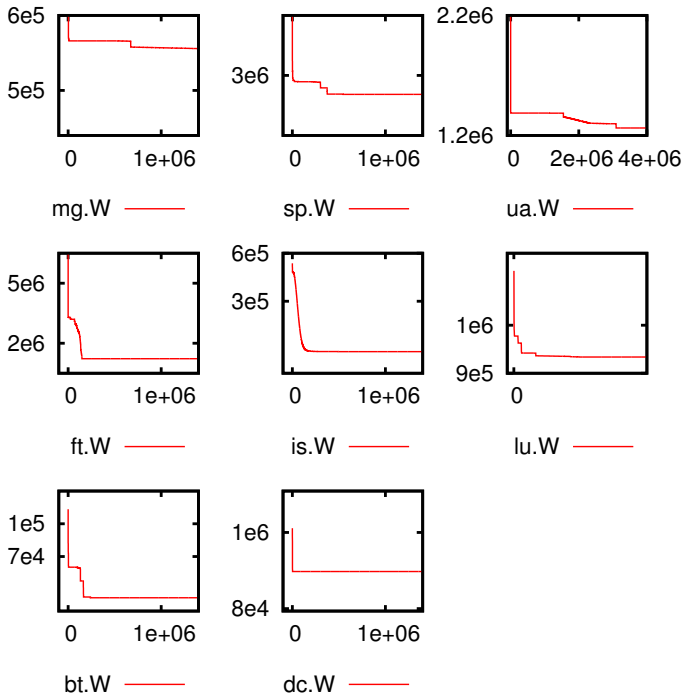

Figure 4.2: MRCs for the NAS benchmark applications.

4.2.2 Generalized ERSS Tree

So far, we have introduced the parameters that describe the memory requirement of an application in a single phase. We now present the Generalized ERSS Tree Model that characterizes all the phases of an application. The Generalized $ERSS$ Tree of an application is a tree structure, where each phase is represented as a node specified by its duration (θ) and $ERSS(\Delta_M)$ function. The phase duration is defined in terms of number of virtual memory accesses and is thus platform-independent. If the $ERSS(\Delta_M)$ function has a sharp knee, we replace the function with a default $ERSS$ value. Smaller phases contained within a larger phase are captured by a parent-child relationship in the tree. Further, if a small phase occurs multiple times within a larger phase, the edge weight between the two nodes represent the number of small phases. Finally, since a single phase may contain multiple phases with

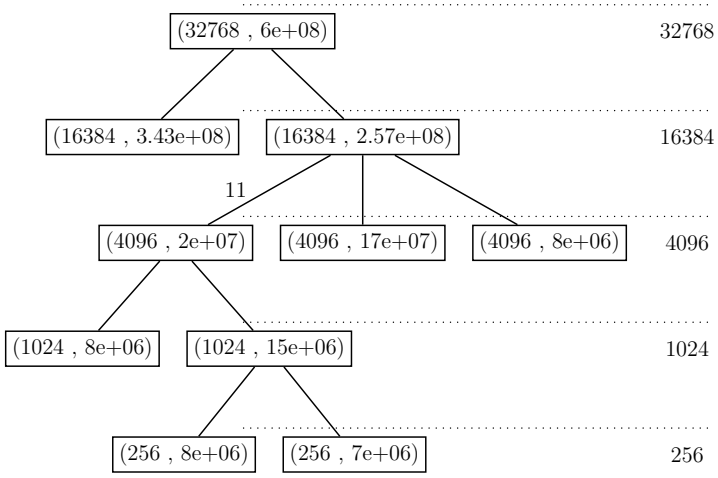

Figure 4.3: A sample Generalized ERSS Tree.

different characteristics, a node may have multiple children.

An example $ERSS$ tree is shown in Figure 4.3 that describes the resource usage of the *bt* application in the NAS benchmark suite. Each node represents a phase with two parameters ($ERSS$,θ). The tree contains 5 levels of phases with the largest phase having a length of 6×10^8 memory accesses and containing two smaller phases, each with an $ERSS$ of $16MB$. The first phase has a length of 3.43×10^8 memory accesses and the second phase has a length of 2.57×10^8. The second phase has three embedded phases of $4MB$ each, where the first child phase repeats 11 times. Given such a tree, one can easily identify the phases that would be resident in any level of memory. A typical example of resident locations of the phases at various levels of the memory hierarchy is shown using dotted lines in the figure.

4.2.3 Wastage

Sharing introduces another dimension of complexity to the resource usage behavior of applications. Since commodity systems do not support strict isolation of various cache resources across applications, the effective cache size (at any level of the cache

hierarchy) available to an application is directly influenced by the resident sizes and rates of accesses of co-located applications.

A common method to estimate reuse set sizes (RSS) is using the miss rate curve (MRC) which characterizes the cache miss rate that the workload would experience at various cache sizes (see Figure 4.4). If we take one of the MRCs on the figure and follow the curve from left to right, the RSS is the size at which the curve becomes completely horizontal: any increase in cache size is futile when attempting to decrease misses. Given the MRC, the cache can be provisioned to match the reuse set size. This works well when a single workload uses the cache exclusively. However, when multiple workloads share a storage cache, this approach does not usually work.

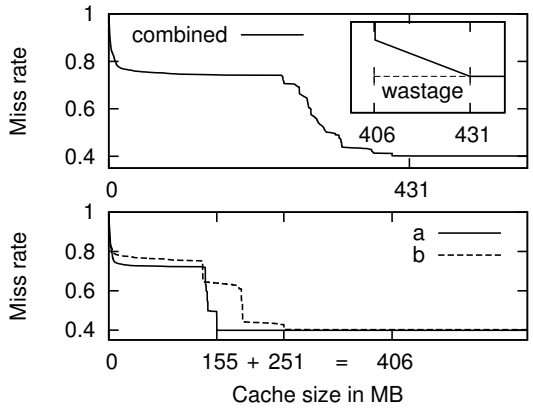

Figure 4.4: Working set sizes and wastage on multi-workload MRC. Curves a and b are the MRCs of the individual workloads and combined is the MRC of both a and b running together. The reuse set sizes (RSS) of a, b and combined are 155, 251 and 431 respectively. Notice how the RSS of combined is larger than the sum of a and b's RSS.

Figure 4.4 illustrates how using individual workload MRCs is not adequate for computing the combined cache requirement of two workloads. We ran two web server traces from a university CS department for a day on a aging-LFU cache simulator. We ran the two workloads first individually to obtain their reuse set sizes (RSS); we

did the same next but with the two workloads running at the same time to obtain the RSS of the combined workload. We would expect the combined workload to have a RSS equal to the sum of both individual RSS. However, the real RSS is 25MB more than the expected 406MB. We term this additional cache requirement on reuse set sizes as *wastage*.

Several researchers have observed this problem for Least Recently Used (LRU) caches [TSW92a, QP06a, STW92, KVR10] when two or more applications are sharing a cache. The common observation they made was that a cache is not guaranteed to entirely host the reuse sets of two applications even when the sum of the individual reuse sets sizes are less than the cache size. This observation also triggered the division of Level 1 CPU caches into instructions and data for most architectures [STW92, Int09].

To quantify the extent of wastage with many workloads, we replayed combinations of I/O workloads from a set of eight actively used systems including 2 production web servers, 5 student desktops and a production University CS department email server. Workload durations were three hours and combination workloads were created by merging based on timestamps, an approximation of how the I/O scheduler would combine the I/O streams before dispatching to the backing store. The combined workloads were replayed on simulators using both LRU (Least Recently Used) and an aging-LFU (Least Frequently Used) replacement policies. We used two migration policies for each replacement: on-demand migrations where a block is moved to the cache just after a miss, and periodic migrations where blocks are moved to the cache every 5 minutes. On-demand migrations are necessary for CPU and buffer caches as blocks must ultimately be used from the cache. On the other hand, SSD caches can be populated periodically, and most caching systems or first level tiers [GPG+11] do so in order to not interfere with the foreground I/O

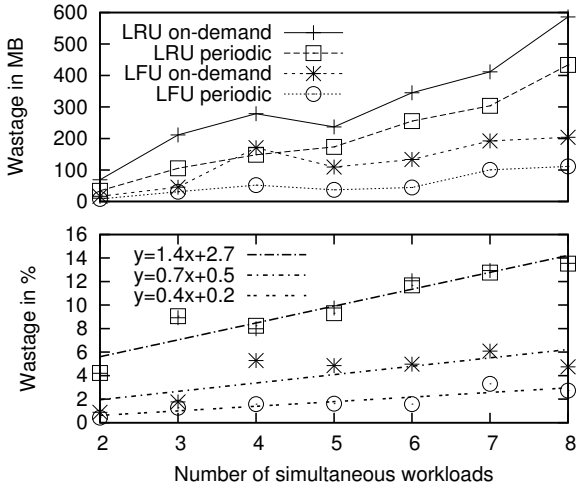

Figure 4.5: Wastage with multiple workloads. Figure on top shows wastage in MB and figure at the bottom shows wastage as a percentage of the total amount of cache: 100*W/RSS_c, where W is wastage and RSS_c is the reuse set size of the combined workload. Notice how wastage as a percentage increases linearly with the number of workloads

workload.

Figure 4.5 shows the average wastage induced per workload as we increase the number of traces sharing the cache. For all cases, the percentage of per-workload induced wastage increases linearly with the number of workloads. Cache block size for all experiments was 4KB; we saw the same effect for blocks of 128KB. It is important to notice that we are unaware of any previous research that noticed cache wastage for cache policies other than LRU.

We now extend our model to account for resource competition. In other words, we estimate application memory resource usage with multiple co-located applications, explicitly accounting for wastage.

4.2.4 Using the Generalized ERSS Tree Model for Provisioning

We wrap up this section with a discussion of how to utilize the model described so far when provisioning resources for a set of consolidated applications. The cache

provisioning approaches in the literature today all suggest allocating an amount equal to the sum of the working sets of each application. The proposed $ERSS$ metric leads to a more accurate estimate of the cache requirements of an application but yet assumes that the application is running in isolation. To address this gap, we modeled the impact of contention due to co-located applications with the additional parameter of $Wastage$ that enable us to perform cache provisioning more reliably.

If we assume an LRU-based eviction policy (typically employed in caches today), then applications memory lines can get evicted by applications that are not reusing their elements. If an application access a memory line, it will get to the top of the LRU stack, as it is the most recent line. This happens even if the line is not reused again. The problem is that this line can evict the line of a second application that will actually be reused: a useful line is evicted by a useless line. This is more likely to occur when an application accesses these useless lines faster than the second application reuses its lines. We define the rate at which an application i reuses its lines as Reuse Rate ρR_I, and the rate at which an application j accesses lines that will not be reused as Flood Rate ρF_j.

Definition 5 Reuse Rate for an application i during a phase. We define reuse rate as $\rho R_i = R_i/n$, where R_i is the number of reuses performed by application i and n is the total number of accesses by all applications during the phase.

Definition 6 Flood Rate for an application i during a phase. We define flood rate as $\rho F_i = F_i/n$, where F_i is the number of accesses to lines that are not going to be reused during the phase by an application i, and n is the total number of accesses by all applications during the phase.

During a phase, we define the wastage that an application i flooding a cache incurs on an application j reusing elements as $W_{i \to j}$.

Definition 7 Wastage from application i to j during a phase. We define wastage as:

$$\boxed{W_{i \to j} = \lceil \frac{ERSS_j * \rho F_i}{\rho F_i + \rho R_j} \rceil} \tag{4.1}$$

when $i! = j$ and 0 otherwise.

Wastage has an cumulative effect: an application flooding the cache affects all other applications sharing it. We observed this effect in figure 4.5.

Definition 8 Wastage: A set of N applications sharing a cache incur wastage on each other as:

$$\boxed{W = \sum_{i=1}^{N} \sum_{j=1}^{N} W_{i \to j}} \tag{4.2}$$

Here we make a simplifying assumption that wastage can be characterized by the sum of pairwise effects. In reality, arbitrary sets of applications can jointly impact each other in terms of cache usage. We now define the isolation condition for a set of applications sharing a cache.

Definition 9 Isolation Condition: A set of N applications A_i sharing a cache of capacity C are said to satisfy the isolation condition iff the following condition holds:

$$\boxed{\sum_{i=1}^{N} ERSS_i + W \leq C} \tag{4.3}$$

Before consolidating, it is critical that the isolation condition holds at various levels of the memory hierarchy to ensure performance isolation. To determine if this is true, the $ERSS$ tree for each application must first be created. All phases that

were resident in a specific level of the memory hierarchy when the application ran stand-alone is determined. Consequently, the phase, termed P_{big}, with the largest $ERSS$ for each application and whose duration is above a pre-determined threshold, is identified. These are the phases for which we would need to ensure the isolation condition (Equations 4.3) must be met. This process is then applied to each level of the cache hierarchy for a given system to conclusively establish if the applications can be consolidated on the given hardware. Finally, in applications with hierarchical phases (smaller phases embedded within larger ones), the hit rate and $ERSS$ for larger phase includes access to memory lines of the smaller phase as well. However, an accurate characterization of the larger phase should be made independent of the accesses to any embedded smaller phases. Hence, the *ERSS* and *Hit Rate* is calculated as the marginal (or additional) $ERSS$ and *Hit Rate* respectively over the smaller phase.

4.3 Building the Generalized ERSS Tree Model

We now present a methodology to construct a generalized ERSS tree (henceforth referred to as ERSS tree), representing an instantiation of the Generalized ERSS Tree Model for the application under consideration.

4.3.1 Methodology Overview

The Generalized ERSS Tree Model is a recursive model with a subtree composed of smaller phases embedded within a larger phase. Our methodology to create the ERSS tree is based on the observation that one can create a coarse tree (with few levels) and increase the resolution of the tree by identifying phases for additional resource sizes iteratively. We start with the root node of the tree which corresponds

to the largest unit in the memory hierarchy (typically disk-based storage) and use a three-step *node generation process* to identify and incorporate phases of the longest duration. We then recursively refine it to include phases at lower levels of the tree by iteratively applying the node generation process for decreasing sizes of memory. To deal with noise or data-dependent execution, the process is repeated for multiple runs and only the phases that are identified in all runs are included in the model. The *node generation process* consists of

1. Refinement Level Identification. In this step, we identify the next size of memory resource, $Avail_{mem}$, (e.g, 2 MB) using which the tree should be refined further.

2. Resource Limited Execution. In this step, we identify the memory resource R that best matches $Avail_{mem}$ and ensure that the application executes with only $Avail_{mem}$ amount of R available. The techniques to ensure this reservation depends on the memory resource R and is detailed in Sec. 4.3.3. This step also creates an *execution trace* including the hits and misses for application requests to the specific resource, which serves as the input to *Atomic Refinement* step.

3. Atomic Refinement. This step uses the execution trace following the *Resource Limited Execution* to refine the ERSS tree. Atomic refinement is described in detail in Section 4.3.4.

It is straightforward to use the ERSS tree for an application generated using the above node generation process iteratively in a consolidation scenario. Prior to consolidation, we refine the tree to closely reflect the proposed allocation of various memory resources for each application to determine actual ERSS sizes at various levels of the memory hierarchy. We then use Equation 4.3 to determine if the planned allocations are adequate or over-provisioned for the dominant phases at each memory level. Next, we elaborate on the steps of the node generation process.

4.3.2 Refinement Level Identification

The *node generation process* must be applied for various levels (sizes) of memory resources in order to create the ERSS tree. A significant advantage of the three step process is the complete flexibility in refining the ERSS tree at the required granularity. The refinement level identification step allows a model that focuses on interesting parts of the tree to create higher resolution sub-trees. Selection of the refinement level in this step would ideally be based on the level of provisioning granularity required and candidate allocation sizes for each memory resource. Thus, the methodology allows easy refinement of the tree for the range of the actual memory assignments (e.g., at all candidate sizes for the L1, L2, and L3 caches and main memory).

4.3.3 Resource Limited Execution

Resource limited execution facilitates application execution for a specific size of the resource under consideration and records the memory hit and miss counters for the application. For example, to identify the memory phases at $ERSS = 2MB$ on a machine with 64KB L1 and 4MB L2, a user would run the application with a resource limited L2 cache size of $2MB$ and measure the L2 hit and miss rates. The hit and miss rate information is available by default on most commodity platforms for all levels of the memory hierarchy. We now present techniques for resource limited execution of the application for a user-specified size limit for various memory levels.

External Memory. The external memory phases of an application form the highest level of the ERSS tree. In legacy systems, the external memory data for an application is different from the other levels of memory in the sense that there are no resource miss events at this level. Consequently, phases and ERSS descriptions in

legacy systems are inconsequential for application performance. On the other hand, these considerations are relevant for systems which employ an external memory device as a cache. Examples of such systems abound in the literature including performance-improving caches [AS95, BGU+09] and energy-saving caches [krm08, UGB+08]. Such external memory devices are typically block devices (e.g., disk drives or solid-state drives). Controlling the size of a block device cache for resource limited execution is achieved easily with block device partitioning. Block I/O tracing tools are available in most commodity operating systems (e.g., Linux blktrace [Axb07]) which work irrespective of the type of the underlying block device. These tools support execution tracing at the partition granularity to ensure non-interference with other block I/O operations in the system. By associating the cache partition block accesses with hits and non-cache partition block accesses to misses, hit and miss events can be recorded.

Main Memory. Techniques for limiting the main memory (henceforth RAM) available to the entire operating system to a specific fraction of the physical RAM exist in many systems. In AIX, we use the *rmss* command to ensure that the operating system can use only the specified size (fraction) of the total physical memory. Once we ensure memory reservation, we run the application and log the page fault rate through the lifetime of the application.

The *Atomic Refinement* step of the node generation process optionally uses the hit rate for higher accuracy. The hit rate for memory pages can be estimated using a binary rewriting API like Dyninst [Ope]. The Dyninst API can attach itself to an executing binary and dynamically instrument the application to get the complete memory trace. An alternative approach to obtain RAM hit rates is to use a full system simulator (e.g., Mambo, QEMU, etc.) and run the application with different RAM configurations.

Processor Caches (L1 / L2 / L3). Various techniques have been proposed to partition the L2 cache at both hardware and software levels. These techniques, however, are not available in commodity systems and implementing these intrusive changes on a production server is not always feasible. The lack of flexible user-level partitioning schemes and fine-grained cache line monitoring counters creates significant challenges in further refining the models for cache resident phases in a non-intrusive fashion.

We developed and implemented two new techniques to partition the cache and record the hit and miss events. Our techniques are accurate for L2 and L3 caches and work with limited accuracy for L1 cache as well. The first technique uses ideas from page-coloring typically employed by the operating system [LLD+08, STS08] to ensure that the application uses only a fixed amount of cache. However, this technique requires application re-compilation. Our second technique, named *CacheGrabber* runs a probe process on the server that continuously utilizes a fixed amount of cache resources, *reusing* the utilized cache in a manner such that it becomes unavailable to the application being characterized. In addition to these, we directly implement two previously proposed techniques for inferring the miss rate curve (MRC) for caches. The first technique uses a cache simulator to simulate caches of different sizes which directly leads to the MRC. The second technique uses sampling of performance monitoring unit (PMU) registers to create a memory trace and replays the trace through a LRU stack simulator to infer the MRC [TASS09]. Further details on these techniques are beyond the scope of this thesis.

4.3.4 Atomic Refinement

The *atomic refinement* step uses the hit and miss events in the execution trace to refine the ERSS tree for that level of memory. If hit rate information is not available

for a memory level, only the miss rate information is used for the refinement. There are two components to atomic refinement at a high level: (i) it detects phases and phase transitions, size estimates, and the instruction time-line during which they occur and (ii) it uses these phases for ERSS *tree refinement*. This ability to detect phases with only the hit/miss information at various levels of the memory hierarchy makes model instantiation feasible in real data center environments.

Phase Detection

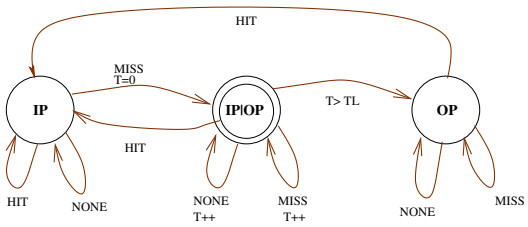

Figure 4.6: Phase Identification State Diagram.

We model the phase transition behavior of an application at a specific memory level using a state diagram, where transitions are triggered by the hit/miss events. The states and the transitions identify various phases and their durations. We model the following states:

1. InPhase (**IP**): The state denotes that the application is within a phase that fits in the memory resource under consideration.
2. OutPhase (**OP**): The state denotes that the application is within a phase that does not fit in the memory resource under consideration.
3. InPhase or OutPhase (**IP|OP**): The state denotes that it is unclear if the application is either in an InPhase or an OutPhase.

The HIT, MISS, and NONE events are *compound* rather than individual events. For instance, a HIT event requires the next window of events exceeds a pre-specified

threshold percentage of hits, rather than a single hit event; if hit events are unavailable, then the we conversely use the requirement that the percentage of misses should be below a certain threshold. A MISS event is defined similarly. A NONE event is one which is neither a HIT nor MISS.

The thresholds used in the above definitions can be computed from the miss rate trace (typically available for all memory levels) for the application as follows. Using the histogram of the miss rates in the trace, we define the lower threshold at the 10^{th} percentile and the upper threshold at the 90^{th} percentile. Due to the steep nature of $ERSS(\Delta_M)$ (Figure 4.2), such a thresholding is sufficient to identify if the phase fits in the available memory resource or not.

The starting state is $(IP|OP)$, indicating that the initial state is unknown. A HIT event in $(IP|OP)$ indicates the start of a phase that fits in the available memory resource and we transition to (IP) state. The absence of a HIT indicates either a *phase transition* or an OutPhase state. We distinguish between the two by using a threshold TL to limit the maximum length of a phase transition. If the time T spent in $(IP|OP)$ exceeds TL, an OutPhase is detected marked by a transition to (OP) state. A MISS event from (IP) indicates a phase transition $(IP|OP)$ and a HIT event in (OP) indicates the start of a phase that fits in memory and is captured by a transition to (IP). The detailed state transition diagram is described in Figure 4.6.

Tree refinement. The *Tree refinement* step uses the node generation process to add new levels in the $ERSS$ tree and/or refine the existing nodes of the tree. The choice of the level to refine is closely determined by the target architecture and the sizes of the various hardware resources in the memory hierarchy. Let $\mathcal{M}_i$ be the memory level currently being investigated and let $ERSS_{\mathcal{P}}$ be the ERSS for a parent phase $\mathcal{P}$, *s.t.* $ERSS_{\mathcal{P}} > \mathcal{M}_i$ The goal is to determine sub-phases within $\mathcal{P}$ that may potentially be impacted by the memory level $\mathcal{M}_i$. If phase detection within $\mathcal{P}$ for

memory level $\mathcal{M}_i$ leads to more than one child phase (IP or OP), a child node for each child phase is added to the parent node. Each InPhase (IP) fits in $\mathcal{M}_i$ memory and is marked with an $ERSS$ of $\mathcal{M}_i$ in the child node, whereas the $ERSS$ of each OutPhase (OP) node is set to the parent memory value $\mathcal{M}_{i-1}$ (as it does not fit in the memory level $\mathcal{M}_i$). If the node generation process with memory level $\mathcal{M}_i$ does not identify more than one phase, the parent phase is refined in the following manner. If the number of misses in the parent phase is found to be equal to those observed with $\mathcal{M}_i$, we refine the ERSS value at the parent node as $\mathcal{M}_i$. However, if the number of misses are larger at the memory value $\mathcal{M}_i$, we retain the earlier $ERSS_{\mathcal{P}}$ value for the parent node.

4.4 Experimental Validation of the Model

We now evaluate the need and accuracy of Generalized ERSS Tree Model. In particular, we address the following questions.

1. What is the need for a unified ERSS tree model?
2. How do reuse rate and flood rate impact memory provisioning?
3. Is the model sufficiently accurate to ensure isolation for consolidated workloads?

4.4.1 Experimental Setup

We used three different experimental testbeds in our evaluation. Our first test-bed was an IBM JS22 Bladecenter cluster with 4 3.2 GHz processors and 8 GB RAM with $4MB$ L2 cache. The experiments conducted on this test-bed used the L2 hit/miss counters available on the system. Our second test-bed was the QEMU full system emulator. QEMU runs memory accessing instructions natively on the host

via dynamic binary translation. We modified the software-MMU version of QEMU and inserted tracing code at binary translation points. Specifically, each time a *translation block* (i.e., binary blocks between jumps and jump return points) is sent to the inline compiler, we insert code for recording every load and store instruction. Since the addresses used by loads and stores may be unknown at translation block compile time, the appropriate register values are recorded at run time. Timestamp information is collected from the tsc register in $x86$ and $IA64$. We ran Linux on the modified QEMU emulator configured with 1.8GB of physical memory with several workloads. The emulator itself ran on a 2.93 GHz Intel Xeon X7350 processor. The traces were then fed into a LRU stack simulator to build the MRCs. As our final test-bed, we used Valgrind to perform fine-grained analysis on single application instances and their resource usage behavior across the memory hierarchy, primarily for generating data to support our approach leading to the Generalized ERSS Tree Model.

Two sets of workloads were used in conducting the evaluation. The first set were the *daxpy* and *dcopy* benchmarks from the Basic Linear Algebra Programs (BLAS) library [BDD+02], which represent building blocks for basic matrix and vector operations. We modified these benchmarks to control the size of memory used, the number of iterations, and injected appropriate idle periods to programmatically control memory access rates. These benchmarks mimic behavior common in many high-performance, scientific computing workloads and the fine-grained control allowed us to experiment with a wide variety of memory reuse and flood rates. Our second set of workloads are from the NAS Parallel Benchmark [NAS]. The NAS benchmarks mimic a wide spectrum of representative application behaviors, from CPU intensive to memory intensive. It also includes benchmarks that represent computational science applications (e.g., *bt* captures the basic calculation of

Computational Fluid Dynamics).

4.4.2 The need for the Generalized ERSS Tree Model

We start by motivating the need for a model such as the Generalized ERSS Tree Model to completely describe the resource usage characteristics of applications, including the need for the ERSS metric for characterizing phases and the hierarchical ERSS tree model for characterizing the overall resource usage of an application.

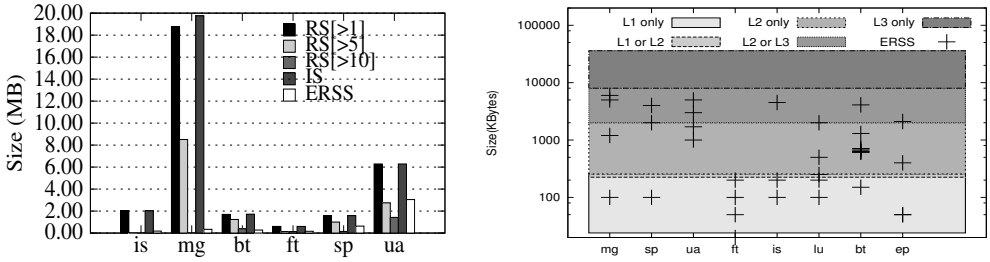

Figure 4.7: (a) IS, RS, ERSS for one phase. (b) ERSS of various phases for NAS applications.

Need for the ERSS metric: We first examine the need for the new ERSS metric introduced in this work for describing working set sizes within a single phase. Specifically, we study if the default ERSS for a phase can be inferred from known measures such as the Inuse Set (IS) or the Reuse Set (RS). We refine the reuse set to require at least a specified number (k) of reuses in a phase, allowing us to define multiple notions of reuse denoted as $RS[> k]$. We used the Valgrind test-bed and the NAS applications to characterize the resource usage behavior within phases in real applications. Figure 4.7(a) depicts the IS, RS and $ERSS$ for a randomly chosen phase within NAS benchmark applications. It is evident that the $ERSS$ metric is distinct from all the alternative measures considered, including variants of reuse set. Since the $ERSS$ characterizes application resource usage more accurately

than either the IS or RS, it is evident that an accurate memory model should be based on $ERSS$ and new techniques should be designed to accurately characterize it.

Need for an ERSS tree: Resources at various levels of the memory hierarchy on stand-alone servers typically have disjoint ranges and provisioning can be handled independently for each resource. First, we show that in production, virtualized data center settings, depending on the type and number of co-located VMs, the amount of resource at individual levels that would be available to a single VM may no longer fall into pre-specified ranges. Further, we demonstrate that a single application may have multiple phases and the $ERSS$ values for these phases may span across multiple levels of the memory hierarchy.

To accomplish the above, we used the Valgrind test-bed to compute the $ERSS$ values for all distinct phases of the NAS benchmarks. Further, to get an estimate on the range of memory resources available to individual applications, we examined the configuration data sheet of a production data center with virtualized servers hosting one or more VMs. Based on the placement per the configuration data sheet, we divided the L1 and L2 cache resources to each VM in proportion to the CPU allocation. We take the maximum/minimum L1 (or L2) cache available to any VM across all servers in the data center as an upper/lower limit on the cache that a VM can use. Thus, we noted that as opposed to a fixed cache size on stand-alone servers, the cache sizes on virtualized servers changes significantly based on the number of hosted VMs.

To remove the influence of outlier configurations specific to the data center, we discarded the top 10% VMs with the largest cache sizes. We then plotted the $ERSS$ for various phases of NAS applications to identify the memory resource that they would fit into in Figure 4.7(b). We observed that most workloads encompass more

than one resource level owing to distinct $ERSS$ values within phases of different granularities. Further, the $ERSS$ of some phases have sizes that overlap in the range for two memory resources. Hence, during memory provisioning, if a phase does not fit in a low level cache, it may be possible to provision a higher level cache for the phase. This underscores the importance of a unified model that captures resource usage across all levels for better cache and memory provisioning.

Application	**bt**	**cg**	**ep**	**ft**	**is**	**lu**	**mg**	**sp**	**ua**
#phases	12	4	3	4	4	5	5	4	6
$ERSS_{max}$	4.1MB	7.25MB	2.2MB	8MB	4.5MB	7.25MB	7.8MB	4.1MB	3MB
$ERSS_{min}$	150KB	50 KB	50KB	20KB	100KB	100KB	100KB	100KB	1MB
# large phases	2	2	2	2	2	2	4	4	2

Table 4.2: Distinct, dominant phases for NAS applications.

Need for phase duration: We use the phase duration (θ) parameter in our model to characterize the relative importance of a phase. Table 4.2 presents the total number of phases as well as the phases that are long-lived. We define a phase as long-lived if running the phase located within and outside of its required memory level leads to a difference of at least 10% in total number of misses at that memory level (across all phases). We noted (not shown) that even for applications with a large number of distinct sized phases, we can safely ignore a large number of the phases (e.g., 10 out of 12 phases for *bt* are very small). It is adequate to provision memory resources only for long-lived phases because doing so would lead to little or no impact to application performance.

4.5 Related work

The most popular model for application main memory usage is the classical *working set* model and its refinements [Den80, Den68b]. The working set model is based on the concept of resident set, or the set of active memory pages within an execution

window. Once the resident set for an application is identified, the operating system may allocate memory to the application accordingly. A second popular model for resource usage in the memory hierarchy is the Miss Rate Curve (MRC) model, typically used to allocate processor caches [RSG93, TASS09]. These models capture the memory usage characteristics of an application during a window of execution, termed as a *phase*. Phases are a third important memory resource usage characteristic of applications. Batson and Madison [BM76] define a phase as a maximal interval during which a given set of segments, each referenced at least once, remain on top of the LRU stack. They observe that programs have marked execution phases and there is little correlation between locality sets before and after a transition. Another important observation made by Batson and Madison and corroborated by others is that phases typically form a hierarchy [RSG93, KS00]. Hence, smaller phases with locality subsets are nested inside larger ones. Rothberg *et al.* [RSG93] present such a working set hierarchy for several parallel scientific applications. However, the working set hierarchy does not capture the frequency of each phase and its impact on performance.

In contrast to previous work, we show that the amount of memory resource required by an application is more accurately reflected by a new metric that we introduce, the Effective Reuse Set Size (ERSS). Further, memory provisioning requires the identification of all the phases of an application and their durations to ensure that dominant phases are resident in a sufficiently fast level of the memory hierarchy. An important aspect of memory usage not captured by any existing model is the rate of memory usage by an application. The *Reuse Rate* and the *Flood Rate* significantly determine the amount of memory resource required by an application. When multiple applications share a common memory resource, these rate parameters dictate the amount of memory available to each application. Hence,

memory provisioning in a consolidated virtualized environment requires significant refinements to existing memory usage models for them to be applicable.

4.5.1 Mechanisms to build memory usage models

Several techniques have been proposed to identify the optimal working set during program execution. Carr *et al.* [CH81] combine the local working set with a global clock to design WSClock virtual memory management algorithm. Ferrari *et al.* refine the classical working set to variable interval size, where the size of the interval is controlled using three parameters – the minimum interval size M, the maximum interval size L, and the maximum page fault Q [FY83]. The page-fault frequency (PFF) algorithm [CO72, GF78] uses the PFF concept to determine the resident set size, enlarging it if the PFF is high and shrinking it otherwise. There are two popular approaches to infer the MRC for an application. The first approach creates the MRC statically by running the application multiple times, typically in a simulator with different memory resource allocations [RSG93, KS00, WOT+96]. A second approach uses a memory trace and a LRU stack simulator [KHW91] to infer the MRC. MRCs in filesystem caches have been estimated using ghost buffers [KCK+00, PGG+95]. Dynamically estimating cache MRCs is much more challenging and estimation methods exist only on specific platforms [TASS09, ZPS+04a]. Detecting phase transitions for taking reconfiguration actions has been explored before. The Dynamo system [BDB00] optimizes traces of the program to generate fragments, which are stored in a fragment cache; an increase in rate of fragment formation is used to detect phase transition. Balasubramonian *et al.* [BABD00] use the dynamic count of conditional branches to measure working set changes. Dhodapkar *et al.* compute a working set signature and detect a phase change when the signature changes significantly [DS02].

4.6 Summary

We have presented the Generalized ERSS Tree Model that addresses two significant gaps in our current understanding of application resource usage characteristics for the multi-level memory hierarchy. First, the model characterizes the memory resource usage of an application for its entire lifetime with a high degree of accuracy. We have presented a methodology to build model instances for arbitrary workloads on commodity hardware without making intrusive changes. Second, and more significantly, this model can be utilized for highly accurate provisioning of the memory hierarchy in a shared application environment. Our model comprehensively addresses the resource usage characteristics when competing workloads introduce non-trivial isolation requirements. It does so by explicitly characterizing the complex interaction between the resource reuse rates and flood rates of various workloads at each memory level. We have demonstrated that the model can be used to predict when isolation conditions are not satisfied as well as to determine resource provisioning requirements to ensure application performance isolation in shared environments. Finally, by experimenting with a wide mix of applications, we have established the utility and accuracy of our model in practice. In the next chapter, we pursue a complementary direction to reduce the contention introduced by co-located workloads that applies to partitionable caches.

The work presented in this chapter was reported in PERFORMANCE 2010 [KVR10] and MASCOTS 2011 [KVR11].

CHAPTER 5

PARTITIONING FOR EXTERNAL MEMORY CACHES

In the previous chapter we observed that sharing caches can lead to wasted space and increases linearly with the number of workloads. In this section, we propose some solutions to reduce cache wastage for out of memory caches, specifically flash-based solid state drive (SSD) caches for disk. It is important to notice that while this chapter considers and valuates proposals for flash-based caches, the system proposed would apply to other persistent caching devices with equivalent or better performance characteristics relative to flash.

Previous researchers noticed the occurrence of wastage in CPU and RAM caches and proposed simple cache partitioning techniques in order to minimize it. These methods are based on assumptions made over the replacement algorithm used and the type of accesses. However, these assumptions do not hold for external memory caches, and can lead to large inaccuracies. For instance, external memory caches do not require blocks to be moved to the cache after a miss (on-demand) and many external memory caches move blocks to the caches periodically.

We propose addressing these inaccuracies by not making any assumption about the frequency of the migrations, the replacement algorithm used, nor the shape of the miss rate curve. We propose having an iterative partitioning algorithm using probabilistic searches that is capable of achieving several performance goals like: minimizing overall miss rate, or minimizing the sum of VMs' average latencies.

We implemented a partitioned write-back host-side SSD cache for virtual disks in the VMware ESX hypervisor. To maintain data consistency after crashes and reboots, we keep the disk-to-SSD address mappings persistent and all cache allocations and demotions journaled on the SSD. Cache partitions are managed by a local LRU or LFU replacement policy with all SSD/disk block migrations being

performed periodically. The partitions are resized periodically to adapt to workload and storage changes.

Experimental results show that partitioning an SSD cache can accelerate the boot time of 28 virtual desktops from 39 using a unified LFU cache to 32 seconds on the partitioned case; this improvement was attributed to a reduction in SSD cache wastage. Microbenchmark-based experiments demonstrate that the system is able to balance the average latencies of two mixed read/write workloads with less than 5 % of error. Finally, for a workload consisting of half random reads and writes, the system maintains a crash consistent cache at the cost of adding 100 ms of latency to less than 1 % of the accesses, but without adding any overhead to the average latency.

5.1 Background on cache partitioning

The wastage problem can be addressed with cache partitioning. Most partitioning approaches in the literature are based on *miss rate curves* (MRC), which define the miss rate incurred by a workload for a given cache size. We illustrate the use of miss rate curves to find the allocation that optimizes a specific goal – minimizing the sum of average VM miss rates. Assume there are two VMs randomly reading from files. VM 1 reads a file of 2 GB and VM 2 reads a file of 1 GB. The MRCs of both VMs are showed at Figure 5.1. The miss rate is 1 for a cache size of 0 for both VMs and miss rate is 0 for cache sizes greater than the files read (2 GB for VM 1 and 1 GB for VM 2). For an available 2 GB of cache space, we can try all combinations and find that assigning 1 GB to each VM is the optimal choice. This partitioning results in a total miss rate of $0.5 = 0.5 + 0$ and is depicted as the shadowed regions in Figure 5.1.

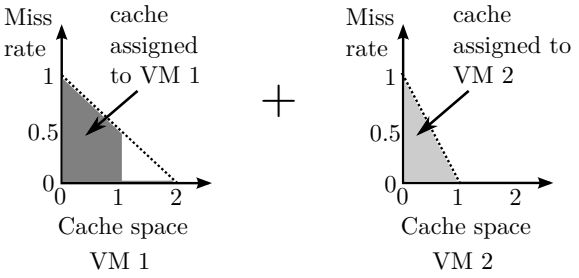

Figure 5.1: Example of cache partitioning. A cache of size 2 is partitioned across two VMs. Both VMs MRCs are shown with the optimal assignment of cache shown in grey: one cache unit to each VM.

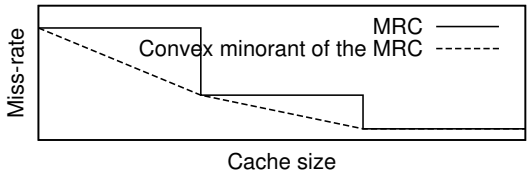

Figure 5.2: Example of MRC and its convex minorant.

While this seems straightforward, in practice, the number of possible partitions grows exponentially with the number of workloads. Rajkumar *et al.* prove that optimal cache partitioning is NP-hard [RLLS]. Therefore, approximation algorithms have been proposed to solve the partitioning problem. Stone *et al.* first proposed using convex hulls as an intermediate step for solving this problem [STW92]. This approach and its variants have been used by several researchers for partitioning CPU and main memory or storage buffer caches since then [GJMT03, QP06b, PGSK09, PSPK09, SLG+09, SRD04, TSW92b, ZPS+04b].

The convex hull approach Current approximation algorithms for cache partitioning largely follow the convex hull approach. The convex hull of a set of points is the smallest convex polygon that contains all the points. The convex minorant is the greatest convex curve lying entirely below this polygon. An example is shown in Figure 5.2. The partitioning algorithm first computes a convex minorant for the

MRC of every VM, and then applies a greedy search algorithm on the minorants to find a close-to-optimal partitioning of cache space as follows:

1. Calculate the convex minorant $m_k(s_k)$ of the MRC for each VM.
2. Initialize partition sizes s_k to zero.
3. Increase by one unit the partition size of the VM which would benefit the most from the increase. The benefit for a VM k is $m_k(s_k) - m_k(s_k + 1)$. Notice that the convex minorant is used to calculate this benefit.
4. Repeat the previous step until all cache space has been assigned.

Problems with the convex hull approach While the above approach works well for CPU and main memory buffer cache partitioning. However, it fails when used for host-side SSD caches and will fail for other out-of-core caches with similar properties. The SSD cache itself is not in the I/O path for all data, unlike CPU and main memory caches, requiring additional work to access items from the cache. Consequently, I/O request sizes that are larger than the cache block size require special consideration. For a multi-block request, some blocks could be resident in the SSD cache while others are not. A request is a cache hit only when all the blocks accessed are in the cache. The effect of this is that for a specific cache size, the chances of missing it increase with the request size. For example, if the cache size is 10 MB, requests of 1 MB are more likely to miss more than requests of 4 KB.

Figure 5.3(a) shows the MRC for a workload that randomly reads a file of one million blocks with a request size of one block. As expected, the miss rate decreases as a straight line from 1 when there is no cache to a miss rate of 0 when the cache has the size of the file. Figure 5.3(b) shows the MRC for the same workload but with request size of 64 blocks. The curve becomes concave because as the size of

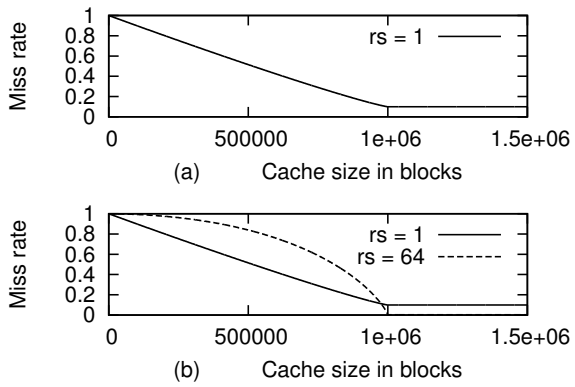

Figure 5.3: MRCs for large requests sizes. (a) shows the MRC of a workload accessing a file of one million blocks with request size of one block and (b) shows the MRC for the same workload and a slightly different one where the request size was increased to 64 blocks. Note how the MRC gets concave with a large request size.

the request increases the chances of hitting the cache decrease. The problem with the convex hull approach is that both these workloads (a) and (b) will have the same convex minorant and therefore will be treated the same way by the allocation algorithm. This can lead to a substantial approximation error. For example, if the workload had requests of size 64 and the algorithm assigned it half a million blocks of cache expecting a miss rate of 0.5, in reality it would incur a miss rate of 0.8, an error of 60%.

In summary, previous research noticed wastage for LRU caches and that the cache partitioning problem needs to be solved with an approximation algorithm. One such approach is using convex hulls as an intermediate step. We showed that wastage occurs irrespective of the cache replacement policy. Further, we demonstrated that the convex hull approach may lead to large errors when applied to SSD caching. In the next section, we outline a new approach that works well with SSD caches.

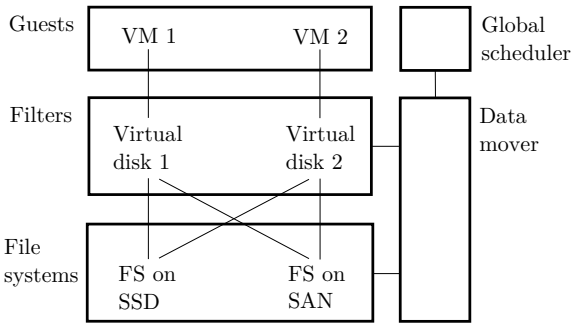

Figure 5.4: Solution architecture.

5.2 Solution overview

Host-side SSDs provide hypervisors a previously unavailable knob for storage performance control. We propose that hypervisors manage this SSD directly as a cache resource to be used within each VM's storage access path. Hypervisors can thereby control the storage performance of individual VMs directly by increasing or decreasing the space made available in this additional cache layer.

Figure 5.4 depicts an architectural overview of the solution. Virtual disks managed by individual VMs are distinct files stored within a file system managed by the hypervisor. The VMs use these virtual disks to store their file systems and swap spaces. In our solution approach, an administrator would assign some or all virtual disks to SSDs for caching and specify goals related to performance maximization and/or QoS on a per-VM basis.

The hypervisor then partitions the cache space across virtual machines in order to meet these goals. Each partition is managed by its own local cache policy (i.e. a cache per partition). Since the SSD is not in the data path for blocks that miss in the SSD cache, unlike conventional demand-based caches, our solution only periodically changes to the set of cached blocks using a *data mover*. Further, unlike previous solutions for CPU caches that continuously partition [TSW92b], SSD

System goal type	Unified cache	Partitioned cache
I. Overall performance	√ (∗)	√
II. Sum of VMs average performance	N/A	√

(∗) incurs wastage because of cache sharing.

Table 5.1: Supported goal types.

cache partitions in our solution are only periodically resized by the *global scheduler* in order to adapt to stable changes in the workloads and the storage system.

We anticipate our solution to be used by administrators towards meeting several types of system goals. Table 5.1 shows the types of goals that an administrator can configure a partitioned SSD cache in our current implementation as opposed to managing the SSD as a unified cache for all VMs. The administrator can choose to *(i)* maximize overall performance which translates to minimize overall cache miss rate or average I/O latency across all VM I/Os or *(ii)* maximize the sum of per-VM average performance which would translate to a fairness goal.

In contrast, a unified cache can support only the first of the two goals above and will incur unwanted cache wastage when doing so.

5.3 The Partitioning Algorithm

Every epoch, the SSD cache partitions are resized in order to achieve some goal defined by the administrator (see Table 5.1). In this section, we describe the process for meeting goal types I and II.

5.3.1 Partitioning for latency minimization

We predict the sum of latencies for a set of potential partitionings and then use the one with the smallest sum using the following steps:

1. Construct an Miss Rate Curve (MRC) for each VM.

2. Use the MRCs to predict latency for all VMs at all possible cache sizes and refer to this as the *latency curve* for the VM.

3. Use a probabilistic search on the latency curves to find the partitioning candidate that produces the smallest sum of VM's latencies.

Step 1: MRC construction for SSD caches. We construct the MRCs using the I/O accesses of the previous partitioning epoch. Based on Denning's locality principle [Den06] it is very likely that the MRCs of two consecutive partitioning epochs are similar. Our MRC construction is an adaptation of Mattson's stack algorithm [MGST70b] to account for the unique characteristics of out-of-core SSD caches including periodic cache allocations (or migrations) and requests larger than the cache block size. Periodic migrations tend to move the MRCs up and requests larger than the cache size should give the MRC a concave shape (as seen in Figure 5.3). The problem is that Mattson's algorithm does not account for these effects.

Mattson's stack algorithm was originally designed to create MRCs for the LRU cache policy; however, it can be used for other policies (e.g., LFU [BEP11]). For every access, the algorithm emulates LRU aging and maintains a histogram of stack distances. This histogram maintains the hit count at every possible stack distance, which is then traversed to create the MRC. Our first modification to the Mattson's stack algorithm is at the histogram maintenance step to record a hit only if all the blocks of a request are in the cache. If the request has more than one block, we update the histogram only for the block that has the largest stack distance. The second modification is done in order to account for periodic migrations. The basic idea is to maintain the histogram of stack distances for an LRU stack whose blocks are only updated periodically (and not after every access). More specifically, we use 2 LRU stacks where the first LRU stack maintains the histogram and the second

implements aging. We update the blocks of the first stack with the second every period in order to mimic the effect of a periodic migration. Finally, the MRC is calculated by traversing the histogram of stack distances.

Step 2: Latency curves construction. We construct a latency curves for VMs using MRCs computed using our modified Mattson's stack algorithm. Latency prediction can be done similar to the approach of Soundararajan *et al.* [SLG+09]):

$$Lat = MR * Lat_{SSD} + (1 - MR) * Lat_{disk} \tag{5.1}$$

where Lat is the predicted average latency, MR is the miss-rate as obtained from the MRC, and Lat_{SSD} and Lat_{disk} are the VMs recently measured latencies for SSD and disk. However, using previously measured latencies can lead to large inaccuracies. A simple scenario where this can happen is when the prediction leads to IOPS saturation on the disk, while the measurements are from an unsaturated state. For example, if the current state Lat_{disk} is low, then a prediction of latency at higher miss rates would fail because Lat_{disk} would not be low anymore.

We handle the latency prediction inaccuracy by iteratively applying our algorithm until we converge to an equilibrium state. The system (1) plans a partitioning, (2) resizes the caches, (3) measurse the resulting latencies, and restarts the whole process again. We stop this process when the deviation between the measured and predicted latencies converge to a difference less than a small constant.

Step 3: Probabilistic search. Once the latency curves have been created, we use a probabilistic search to evaluate potential allocations. These allocations are evaluated by adding up the latencies for all VMs at the potential partition sizes. We chose simulated annealing (SA) [KGV83] as the probabilistic search to look for

the best partitioning. While classic hill climbing only allows changing to a better solution, SA probabilistically allows changes that may degrade the solution.

5.3.2 Partitioning for all other performance goals

The previous approach for minimizing sum of VM latencies can be easily adapted to meet other goals such as maximizing overall IOPS performace. For overall IOPS maximization, the only difference is that instead of using latency curves we use IOPS curves. IOPS maximization and sum of VMs averages reduce to the same problem: maximizing the sum of VMs IOPS obtained from the SSD cache. We minimize the inverse of IOPS ($IOPS^{-1}$) and predict IOPS using the following equation:

$$IOPS = \frac{IOPS_{disk}}{MR} \tag{5.2}$$

where MR is miss rate and $IOPS_{disk}$ are the recently measured IOPS of the disk only. This equation uses the observation that each time the disk has an I/O, the SSD has $(1 - MR)/MR$ I/Os. Therefore, the total I/Os in a second are $IOPS_{disk} + IOPS_{disk} * (1 - MR)/MR$.

5.4 Implementation

An implementation of our proposed partitioned cache is integrated into VMware ESX hypervisor. As in many virtualization architectures, all I/Os initiated by virtual machines are handled by the hypervisor. As illustrated in Figure 5.4, the virtual disk layer of the hypervisor translates a guest I/O into a file I/O to be stored in the hypervisor file system on a SAN. Our system intercepts all I/Os at the virtual disk layer using a filter. An instance of the filter is attached to a specific virtual disk and redirects all I/Os to the SSD on a cache hit or to the SAN on a miss.

5.4.1 Cache management

Caches, like virtual disks, are saved as files on a file system on the SSD. We use one cache file per VM and partitioning is implemented by limiting the physical size of these files. To simplify our prototype, we restrict VMs to only one virtual disk and thus a single cache file. These cache files are populated during each "mover epoch" by a data mover process that moves blocks to and from the SSD periodically based on a cache policy. We implemented both LRU and aging-LFU replacement policies for each cache partition. Typical cache replacement policies are designed for CPU and buffer caches and therefore they need to move blocks to the cache directly after a miss. We observed that for all the traces used in the motivation, we can improve overall throughput by making such data movement periodically rather than in the IO path. The global scheduler which is in charge of partitioning the SSD space among the various cache files implements the partitioning algorithms presented in the previous sections. The global scheduler runs every "partitioning epoch", which is typically five minutes.

5.4.2 Data consistency

We implement caches as write-back and special care needs to be taken to ensure data consistency after system crashes and restarts. Write-back caches can contain dirty data not present in the backing store. Therefore, VMs need to be restarted by accessing the backing store in concert with all its cached data on SSD. We keep the mapping information for cached data persistently in the SSD cache itself so that it can be queried after VM crashes and restarts. To minimize the impact on I/O performance, we journal the updates to these persistent mappings in memory and then replay the journal before data mover migrations during each mover epoch.

5.5 Experimental validation

We now present an evaluation of the techniques proposed in this chapter using an implementation in the VMware ESX hypervisor. We first list the questions that we want to answer through experiments:

1. How does a partitioned cache perform compared to a traditional unified cache?
2. What are the performance overheads and where they are coming from?
3. How do these techniques adapt to specific storage characteristics such as slow-writing SSDs?

5.5.1 Experimental Setup

We implemented our SSD caching system as a filter at the virtual disk layer of the VMware ESX hypervisor. The cache block size is 128 KB, partitioning epoch is 5 minutes, and mover epoch is 1 minute long. We use the aging-LFU replacement policy [RD]. The host was an AMAX with a SuperMicro H8DA8 motherboard and two AMD Opteron 270 dual cores processors, each running at 2Ghz and 4GB memory. An Intel X25-M was used as the host-side SSD attached to the host. The backing store was an EMC Clariion CX4 Model 120. The data mover and the hypervisor were configured to have an issue queue length of 32 pending IOs each. Thus, the data mover migrates 32 blocks at a time.

We evaluated the system with a diverse set of workloads running on various VM operating systems to verify robustness under a variety of conditions. We used two kinds of VMs: a Linux Ubuntu VM with a 30 GB virtual disk, one virtual CPU, and 1 GB of memory; and a Windows 7 VMwith 40 GB of virtual disk, one VCPU, and 2 GB of memory. We evaluated using a mix of microbenchmarks and macrobenchmarks. The microbenchmarks used IOmeter [iom] on Windows and

fio [fio] on Linux for workload generation. We configured them to use different IO sizes, percentage of randomness and reads, and the number of concurrent IOs. The IO engine used in fio was libaio. All experiments used 32 concurrent IOs unless stated otherwise. The macrobenchmarks used were Filebench [McD] OLTP and webserver workloads.

5.5.2 Partitioned SSD cache

In our first experiment, we evaluated the effectiveness when maximizing average I/O throughput with the partitioned SSD cache when compared against a unified cache. Both variants used the same implementation with the only difference that in the unified case the partitions do not have local cache policies and are managed by a unified cache policy. We fixed the total cache size in both variants to 6 GB. The partitioned cache was configured to maximize overall IOPS.

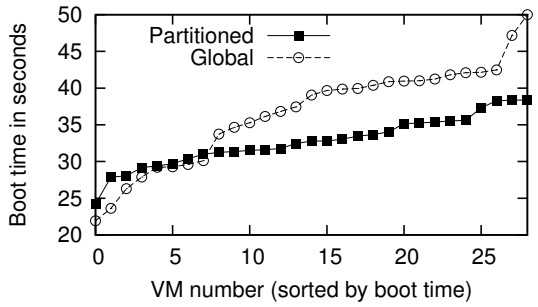

Figure 5.5: 28 Virtual Linux desktops booting simultaneously using an 6 GB SSD cache. This figure shows the boot time using a global LFU cache and a partitioned LFU cache.

We compared both caching options when booting 28 Ubuntu 10.04 desktop systems. The comparison metric is rebooting time as measured by Bootchart [boo]. Rebooting was needed to warm-up the caches. Figure 5.5 shows the booting time of all 28 machines for global and partitioned caches. The x axis is sorted by re-boot time. Average boot time was 32 seconds with our partitioned cache vs. 39 seconds

using a unified cache. We attribute this difference to wastage. By partitioning the cache, wastage is eliminated, and hence the VMs incur less SSD cache misses on SSD, get more IOPS, ultimately improving the boot time. The parititioned cache also reduces the variance in boot time relative to the unified case wherein some VMs boot up to two times faster than others owing to some VMs getting as much as 3 times more cache space than others. On the other hand, the partitioned cache maintained all partition sizes approximately the same. The reason for this difference in variance is that in the unified case, the cache allocations of specific VMs change more rapidly than in the partitioned case, and therefore small changes in its working set leads to changes in cache allocated to a VM.

5.5.3 Overheads

This experiment measures overhead by comparing performance between 3 configurations: (1) our partitioned caching system, (2) when the workload accesses the backing store (SAN) directly without any caching, and (3) a variant of our system where we do not maintain persistent block mappings for the SSD cache.

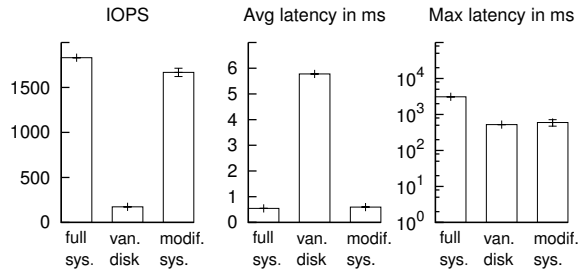

Figure 5.6: System performance compared to vanilla disk and a simplified cache. 4kb aligned random reads/writes on 1GB using 100MB of cache.

We used a single IOmeter workload within a Windows VM for this experiment. The workload is a 50% mix of reads and writes on 1GB with a cache of 100MB. Figure 5.6 shows warm cache IOPS, and average and maximum latency for each of the

three configurations. The first observation is that in the average case, both caching systems perform better than the system without caching. The caching variants both provide approximately the same average latency and IOPS indicating that average case overhead is acceptable. However, the maximum latency of the full system is almost one order of magnitude higher than the one without persistent mappings. We noticed that these spikes in latency occur every mover epoch, and specifically, every time the journal is replayed. The second interesting observation is that the IOPS of the full system are slightly higher than the non-persistent cache variant; it turned out this was because journal replaying was merging and sequentializing writes much better.

5.5.4 Adaptation to specific storage characteristics

Some SSD specific performance oddities are slow writes and sequential read performance being comparable to that of a SAN store. Conventional caches would minimize the number of I/Os going to the backing store, even if it were performing faster. We adapt to such scenarios automatically by eliminating cache space for VMs which do not benefit from SSD performance.

The next experiment mixes one sequential read with a random read workload. The sequential read was setup in a way such that it gets the same IOPS when placed in SAN or in SSD, but gets 8 times more latency when placed on SAN. This table shows the specific performance characteristics of this workload on SAN and SSD:

seq (OIO=8, bs=32k)	Latency (usec)	IOPS
SSD	648	1886
SAN	1548	1689

This sequential workload reads over a file of 5 GB and the random reader over a file of 10 GB. The latency curves are showed on Figure 5.7(a). Based on these

curves, in order to minimize average latency the best allocation is giving half of the cache to each VM.

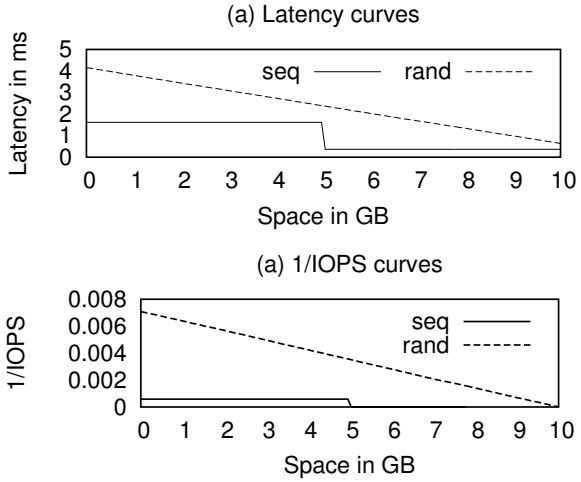

Figure 5.7: Latency and IOPS^{-1} curves.

The best allocation for minimizing latency is half cache to each VM; however, maximizing IOPS leads to entirely different allocations because the sequential workload gets the same IOPS from the SSD and from the RAID. Figure 5.7 (b) shows the cost curves and how the best allocation to minimize $1/IOPS$ (i.e. maximize IOPS) is to give all space to the random workload. Thus our solution produces the expected behavior, i.e., not allocate SSD cache space to workloads that do not need it.

5.6 Related work

We subdivide the related literature based on what type of caches are being partitioned. We finalize this section with a description of the related research on storage QOS.

CPU caches. Early work on partitioning shared cache resources was targeted for CPU caches for instruction and data streams. Stone *et al.* [STW92] pointed out that

LRU caches suffer from slow convergence towards optimal allocations when multiple workloads share the cache. They analytically show that for minimizing overall miss rate and assuming convex MRC curves, optimal partitioning when minimizing overall miss rate can be achieved by choosing allocation sizes for which derivatives of miss rate is the same across all workloads. Thiebaut *et al.* [TSW92b] then developed a practical implementation of Stone's algorithm by approximating MRC derivatives rather than constructing full MRCs and then making them monotonic to force convexness of the implied MRC. Unlike our approach of periodic partitioning, they continuously partition the cache using the robinhood approach (of taking from the rich and giving to the poor) until the MRC derivatives equalize to achieve Stone optimality.

Buffer caches. Arguing that pollution for LRU based main memory page replacement cannot avoid cache pollution, Kim *et al.* [KCK+] demonstrated the benefits of partitioning main memory. Their approach detects sequential accesses and then assigns a single page to each sequential stream. MRCs have also been used to partition main memory. Zhou *et al.* [ZPS+04b] argue that most MRCs for main memory workloads are convex and use a greedy MRC-based approach to partitioning main memory for minimizing aggregate miss ratio. Soundararajan *et al.* [SLG+09] address partitioning of multi-level caches including memory and storage caches. Assuming that these caches can be made exclusive by implementing DEMOTE [WW02b], they partition all the cache space (both levels) in order to minimize average latency using latency curves that are very similar to ours. Their search algorithm uses hill climbing starting from a set of k random partitions to find an optimal one. Goyal *et al.* [GJMT03] address the problem of dynamically partitioning storage memory caches according to goals and changes in the workloads. They address both per-

formance maximization and QoS goals for latency only on a workload-class basis. They use an MRC based approach similar to that used for CPU caches combined with a greedy approach for benefit maximization. They also assume that MRCs are convex.

SSD caches. Sehgal et. al. proposed a SSD cache that uses partitioning to limit workloads' average latencies [SVS12]. Their partitioning technique uses a control loop of observing latency, and increase or decrease cache size in small steps. Their feedback controller increases the partition size when the limit is not met and decreases it when observed latency is better than expected. The problem with this approach is that it is not capable to find minimum average latencies, and requires the assumption of convex latency vs. cache sizes curves.

Our approach has the following distinguishing characteristics from all of the related work. We generalize observations on wastage that have been demonstrated exclusively for LRU caches to apply irrespective of the replacement policy. We empirically demonstrate that the assumptions of convex MRCs or even translations to convex minorants do not apply to out-of-core caches; and neither do current MRC construction methods.

Storage QoS. Storage QoS has been addressed from several perspectives ranging from partitioning memory to I/O throttling to storage migration. We can examine QoS controls that are available to the hypervisor to compare how fast they react to a change in the system. First, memory ballooning [Wal02b] can impact the guest file system buffer cache quickly, but it is difficult to exactly control the size of the buffer cache. Second, memory partitioning has been used in order to provide soft-QoS controls for workloads [PSPK09, PGSK09]. Third, I/O throttling [GAW09] allows for rapid control of the VM I/O performance in a cluster. Next, comes I/O

scheduling [GMV10]. Lastly, migrating virtual disks [MCGC11, GKAK10, GAW11] is a heavy weight options that may take many hours to implement.

Host-side SSD caches provide a control knob for storage QoS that is complementary to all of the above solutions. It allows for greater scope for storage caching, owing this to its relatively larger size than main memory ballooning. However, it is less responsive than ballooning. In contrast to the I/O throttling, I/O scheduling controls, and storage migration approaches that are ally ultimately limited by storage performance. SSD caching provides a mechanism that allows greater separation of VM I/O performance from storage performance.

5.7 Summary

Host-side SSDs open up a new spectrum of possibilities for improving and managing storage performance in virtualized environments. In this chapter, we demonstrated that host-side SSDs demand new techniques for performance management that are not immediately obvious and proposed solutions that address both performance maximization and storage QoS goals. Further, when employing cache partitioning, we also demonstrated that conventional partitioning techniques, that have worked well for CPU and main memory and storage buffer caches, do not work for SSD caches; the out-of-band nature of the SSD cache and multi- cache-block request considerations both contribute to invalidating assumptions that have been made regarding cache partitioning. Based on these findings, we proposed a new cache partitioning approach that employs online MRC construction coupled with periodic partitioning and data movement to efficiently allocate cache space between the VMs. Our evaluation of an implementation of our solutions in the VMware ESX hypervisor demonstrated that our solution works well in practice and can be used for meeting both performance maximization and storage QoS goals.

A significant requirement of the solutions proposed in this chapter is that cache performance should be affected by sizing. If this requirement is not met, the algorithms presented can not be used for controlling metrics like latency, as the only knob available is resizing the cache partitions. Unfortunately, this requirement is violated when using write-through caches and the workloads are write-intensive. Since the write-through caching policy treats all write accesses, misses or hits, in the same way, making large caches equal in performance to small ones. For this reason, we used a write-back caching policy for the work in this chapter which allows cache write performance to be affected by sizing. However, write-back caching when used for permanent storage introduces data consistency problems. In the next chapter, we investigate the scope of these problems and introduce new write-back caching policies that provide stronger consistency guarantees.

Limitations. It is important to address the limitations of the evaluation for our partitioning technique. The first limitation is scalability; we tested a consolidation ratio of 28 to 1, however, consolidation ratios of more than 200 to 1 has been reported. The second limitation is related to the migration epochs (5 minutes). We have not tried different epochs and we do not know if a dynamic epoch could perform better.

CHAPTER 6

CONSISTENCY IN WRITE-BACK CACHES

An important design choice about caches is how writes are managed, specifically, whether to maintain dirty copies of the blocks in the cache and how to evict them. Caches that maintain dirty blocks are referred as $write-back$ and caches that do not as $write-through$. In the previous chapter we assumed write-back caches as they are likely to perform better and because the cache resizing technique used to control application latencies may be ineffective with write-through caches. One such case is when the application performs a sequence of synchronous writes. If we were using a write-through cache, the system would not be able to control it's I/O latency in any range. The reason for this is that whether writes are hits or not, the write-through policy treats them in the same way, leading to the same latencies. Ideally, we would use a write policy like write-back which can be affected by cache size, but that can also provide some basic consistency guarantees.

Write-back and write-through are extremes in performance and consistency. write-back aims to provide good write performance by minimizing accesses to slower backend devices, but at the cost of possible inconsistencies. Write-through, on the other hand, only improves read accesses performance, but guarantees consistency of stored data. The basic trade-off is between better write performance against staleness or potential inconsistencies. Staleness is a measure of how far behind in time is the data: for example, yesterday's files in a versioned file system are stale [RT03]. In this chapter, we demonstrate that write-through and write-back are point solutions in a spectrum of solutions which offer trading off performance, consistency, and staleness guarantees. Specifically, we propose new write caching policies that present opportunities to trade off these requirements in a more file grained manner. We design and evaluate these write policies in the context of the system presented

in previous chapter, i.e., host-side SSDs serving as a cache for SAN storage. We use the term SAN storage for any storage that is accessed over a network and is slower than the performance offered by the SSD cache.

6.1 Background

We now study the write-back and write-through policies in more detail. We start by discussing data consistency and staleness. Then, we analyze write-back and write-through in terms of performance, consistency and staleness.

6.1.1 Consistency and staleness

We now explain staleness and each type of consistency when applied to different components. According to the architecture of the system 5.4, there are 3 components that could crash: guest, hypervisor or storage server. These crashes could be the result of memory corruption, software bugs, hardware errors, or power outages. These component crashes can result in host crashes that may need a reboot or render the SSD inaccessible even after reboot. We do not consider storage server side errors as part of this thesis. We assume that these are fault-tolerant storage servers with built-in redundancy and fail-over mechanisms to recover from internal failures.

Consistency. Data consistency refers to the guarantee of reading correctly what was written into a block. This ensures that all data read is not garbage and it belongs to a specific file and not to another file. This data and the related file system metadata are manipulated by file system calls. File systems ensure consistency by making file system updates atomic. For example, a file deletion in UNIX consists of removing the inode from the parent directory and deleting the respective blocks in

the bitmap of free blocks. These two operations need to be part of a single atomic operation. If any of these operations is performed and the other is not, both data structures are inconsistent between each other (i.e. the parent directory points to deleted blocks).

In order to achieve file-system consistency at all times, file systems use journaling [Hag87, Twe98], soft-updates [MG99] or copy-on-write [RO91]. These three techniques achieve this by making file system calls atomic. Journaling does it by writing updates to a journal, applying these updates and only then write a commit block that specifies that the changes are ready. Soft-updates achieves this in an indirect way, it reorders updates so they result in a consistent view of the file-system. Copy-on-write writes all new updates in a different location and atomically (a single write) changes the old to the new view of the file system. The relevant observation is that all of these techniques achieve consistency by carefully ordering the updates to disk.

Staleness. Staleness is a measure of how old the newest available data is. This is a different concept than the creation date of the data. This is an orthogonal concept to consistency since data may be consistent but stale. This idea is used to versioning and asynchronous remote mirroring. Some file systems allow to save point-in-time versions of the files [RT03, Dra, New] called checkpoints. These checkpoints are consistent but old versions of the files. With asynchronous remote mirroring, data in the secondary site may be older than data in the primary site [JVW03].

6.1.2 Write-through and write-back policies

Figure 6.1 shows these policies graphically. The sub-figure in the right shows a typical write-back access. Note how the write is returned right after the write to the SSD, contrasted to the write-through case where there is a write access to the SAN

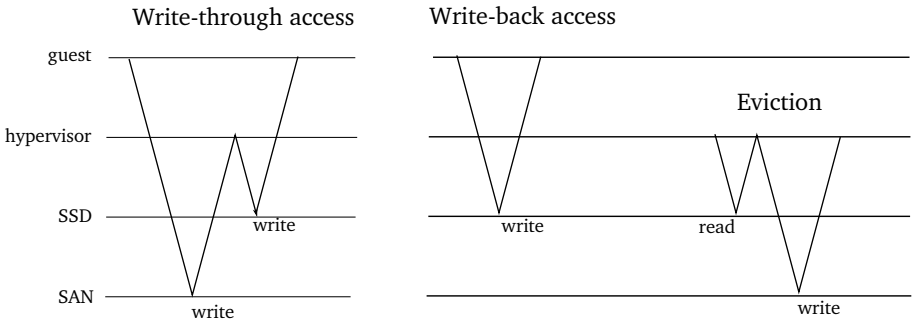

Figure 6.1: Representation of a write access for write-back and write-through write policies.

storage as well. Also, notice how in the write-back case, after some time, there is an eviction of the dirtied block to the SAN storage. This figure shows the performance implication of having one policy versus the other. In this case, the write-back access requires half the time of the write-through case when conservatively assuming the SSD access time to be similar to the SAN storage access time.

A second big difference between these two cache policies is consistency of data in the storage system. File systems rely on ordering of the underlying storage system for its consistency guarantees. The problem with write-back caches is that evictions may occur out-of-order relative to the write sequence of blocks. In case of journaling file systems, this would mean that the commit block may be written prior to the journal entries, resulting in a metadata inconsistency. The write-through policy on the other hand writes to the cache and to the back-end store as a single atomic operation, so they maintain ordering and consistency.

Finally, the third difference is related to staleness of data in the SAN storage. Write-through caches do not introduce staleness, as writes to the SSD are performed immediately after or as a single atomic operation with the write to the SAN storage. The write-back policy on the other hand introduces staleness: writes to the SAN storage are only made by evictions from the cache, which are usually delayed.

	seq write	seq read	random read	random write
OCZ	82	93	177	66
iSCSI	891	593	4813	5285

Table 6.1: Access times in microseconds/access using IOZONE over a OCZ PCI-express SSD and a remote RAID over iSCSI.

6.2 Motivation

We ran benchmark workloads to contrast the performance difference between write-through and write-back caches. These experiments show that there are huge performance opportunities missed when using write-through caches. We ran TPC-C [Tra], an OLTP (Online Transaction Processing) benchmark that simulates a complete compute environment where artificial users perform transactions to a database. According to TPC, this benchmark represents "any industry that must manage, sell, or distribute a product or service".

The illustrative experiment consists of a run of TPC-C with the users, engine, and database in a single machine. We attached a OCZ PCI-express SSD to the machine as host and used a remote storage server over iSCSI [isc04] as back-end store. This remote store uses a RAID5 of 8 7200 RPM disks. Access times are showed in table 6.2. Notice how SSD random writes are 80 times faster than iSCSI accesses.

We then executed TPC-C on 10 warehouses with the two write policies, write-back and write-through, and then measured response time for all transactions. Figure 6.2 shows the median and 90th percentile response time for all 4 types of transactions. All four show the same trend, they have a response time of 2 seconds using a write-through cache and 0.4 using a write-back cache. Having a write-back cache reduces the response time by a factor of 5. This is far from the factor of 80 seconds seen for random writes in Table 6.2, but it is due to the fact that write-back caches

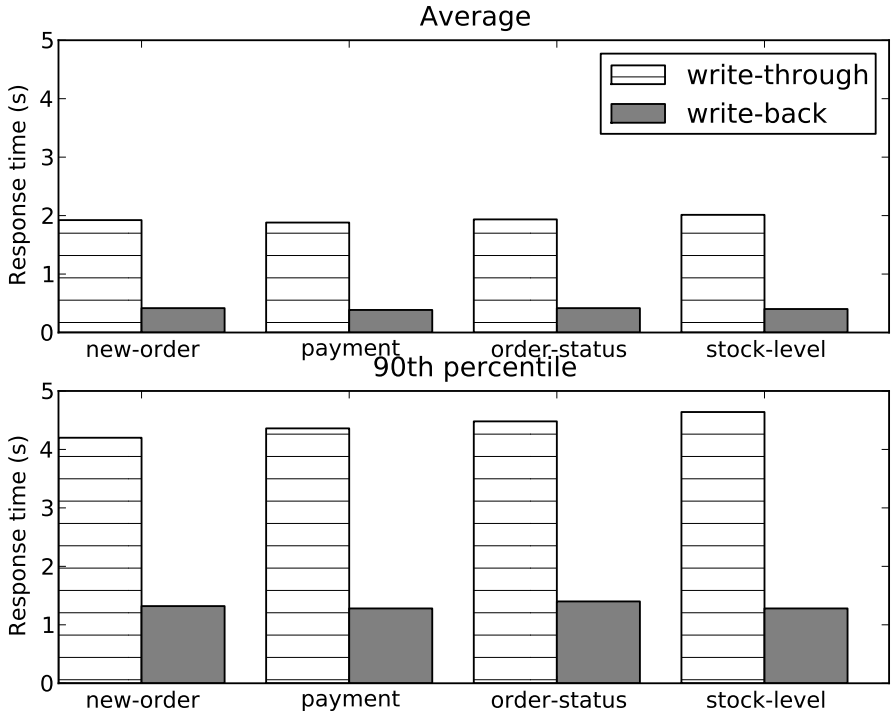

Figure 6.2: Transaction response times for TPC-C using 1GB of RAM, 10 warehouses and 10 clients.

also require evictions, and that transactions have other components that make the overall execution time: reads, processing, and sequential accesses.

Although by having a write-back cache we can reduce response times, it cannot be used in a production environment without fully addressing the data consistency issue that it introduces. The main question we try to answer in the rest of this chapter is whether it is possible to have a write policy capable of achieving better performance (when compared to write-through) that can provide *some* form of consistency.

6.3 Consistency-preserving Write-back Caching

We now present two modified write-back policies that maintain file system consistency after crashes. When using write-back caches, the only way of performing updates to the SAN storage is through evictions. Therefore, if we evict blocks using

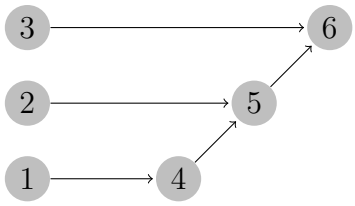

Figure 6.3: Dependency graph. Each node represents a write.

the same order as the original file system writes, we are maintaining the same order for the writes to the SAN storage, thereby ensuring file system consistency. For instance, a file system commit block will only be evicted and therefore written back to the storage after the journal transaction entries have been written back.

The following policies vary in how they maintain the ordering, and how they ensure it when evicting blocks from the cache.

6.3.1 Ordered write-back

To maintain the order of writes for evictions, we need to store the ordering and the data written. The data is stored in the SSD, as new writes do not overwrite old data until it is written back. The ordering is stored using an in-memory data structure with information about where the copies are, and what should be the order to evict them from the SSD.

An intuitive approach to store the ordering is as a list of blocks sorted by completion time. However, with this alternative we would not be able to utilize the fact that some writes can be sent in parallel. Writes can be sent in parallel if for instance there are many applications writing at the same time: there is no dependency between these writes. And, as noted in the previous chapter, I/O parallelism can be a source of good performance. A better alternative is to store the dependencies as a graph, where a block may depend on the completion of many independent blocks. The cache could then evict all the independent blocks in parallel.

The ordered write-back policy maintains ordering and indirectly stores parallelism information using a graph, where every node represents a page write I/O and contains information about the location of the page in the SSD cache as well as its intended final (permanent) location in the SAN storage. An edge from node *dep* to node n represents a completion-issue ordering dependency. This means that the issuing of n occurs after (depends on) the completion of *dep*. As an example, let us assume we have the following sequence of I/O issues and completions: $I_1, I_2, I_3, C_1, I_4, C_2, I_5, C_3$. Figure 6.3.1 shows the related dependency graph. I_i represent an issue event for a write to SAN storage sector i and C_i is the event for completion of the write to sector i.

Whenever we have to evict something from the cache, we first check if it is dirty, in case it is, we have to issue a write (from SSD to SAN storage). But before we do, we need to ensure that any dirty block that this eviction depends on are evicted to SAN storage first.

We now describe how to maintain this graph. Nodes are inserted and modified for I/O issue and completion events. The graph is initialized with an empty set c. We now describe the *issue* operation.

1. Add a node.
2. Add a link for all nodes in the set c to the new node.

We now describe the *completion* operation.

1. Remove from the set c all the nodes that the completed node depends on.
2. Add the completed node to the set c.

Notice that we use the observed completions and issuing of the writes to construct the dependency graph. This may be overestimating the dependencies, that is, we

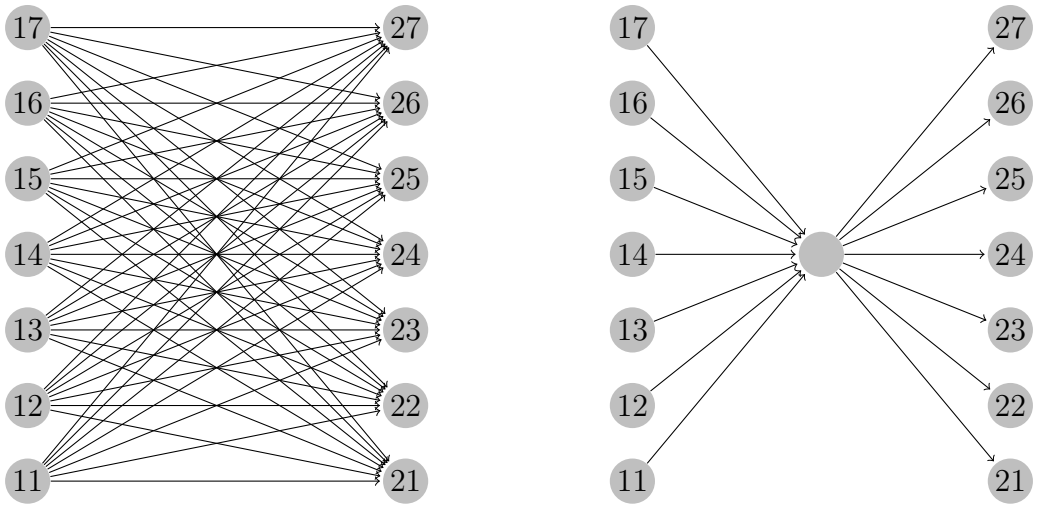

Figure 6.4: Technique used to reduce the amount of memory used for maintaining dependencies.

are maintaining some order that may not be required by the application. If for example 1000 I/Os were meant to be issued independently and in parallel by the application, we could evict all of them in parallel as well. However, if one of them was completed before another one was issued, then we would have to maintain that dependency even though it was not required: the application did not wait for the completion of the first write.

Issue and completion operations require $O(1)$ of computation each, however, memory usage, specially the amount of links can grow up to $\frac{n^2}{4}$ links for a cache size of n pages. This undesirable property was the source of substantial inefficiency in preliminary implementations of this approach. We found a simple and effective optimization to drastically reduce this overhead. We insert dummy nodes after a fixed number of nodes to absorb all the dependencies. Figure 6.4 shows the original graph in the left and the one with the inserted dummy node in the right. Notice how this reduced the number of edges from 49 to 14. Our solution does not explicitly detect high dependency situations like that in Figure 6.4 but rather always inserts a dummy node after every 100 I/O completions. Therefore, if we do not find any

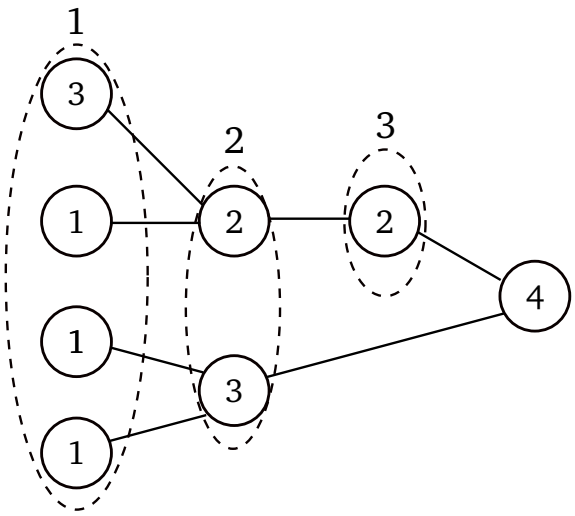

Figure 6.5: Eviction of node 1 requires the parallel eviction of nodes in set 1, then set 2, and finally set 3.

situation like the one in the figure we are wasting 1% of node space, but in the best case we are reducing 100^2 links to only 200. In practice, this heuristic resulted in excellent memory savings for the workloads we evaluated our system with.

Every time some block is evicted from the cache, we need to first evict the dependent blocks in the order prescribed by the dependency graph. We do this by evicting in parallel all the *independent* nodes that the evicted node depends on either directly or indirectly. This would be the set 1 in Figure 6.3.1. We then continue evicting set 2 in parallel, then set 3 and finally we can evict the original node.

6.3.2 Journalined write-back

Ordered write-back had the drawback of requiring to store all copies of the same block in the cache, thus wasting precious cache space. We now present a second write policy for write-back caches where we try to improve the consistency provided by write-back policies. This new technique has better cache utilization and therefore better performance than ordered write-back.

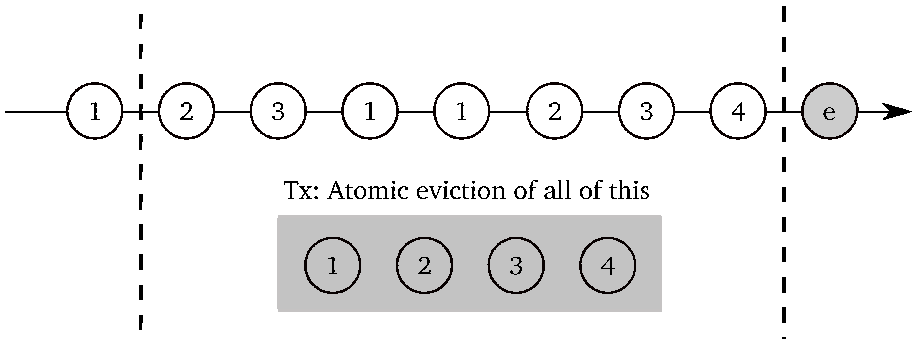

Figure 6.6: Use of a journal for eviction of a list of blocks.

The basic idea is to use journaling in order to provide a safe version of the file system/data. Figure 6.6 shows a list of block writes to cache that occurred before evicting a block e. When using the write-back ordered policy we would have evicted all blocks written before e in order to maintain file system consistency. With this second technique we also maintain ordering but by writing all the previous blocks since the last transaction as a single atomic operation. By doing this we can ensure ordering, because the transaction is executed completely (i.e. there is ordering), or not (i.e. there is also ordering as the previous version was consistent).

Host-side journal. We maintain a simplified journal, with a single transaction in place. The entries for the transaction are a list of blocks to be evicted from the cache. We do not include the data in the entries as the data is available in the SSD. For example, the transaction for the example in Figure 6.6 would just be the list $[1, s_1], [2, s_2], [3, s_3], [4, s_4]$, where s_i is the position of block i in the SSD. The transaction is stored in a journal space in the SSD which can be accessed and replayed in case we crash while performing the transaction. We do not need more than a transaction at a time, because the list of one of the transactions would always be a sub-list of any other transaction.

Storage-side journal. With this approach, the host asks the storage server to evict all the blocks dirtied before e as a single atomic transaction. The storage

server receives a list of block numbers and data to evict, and stores the list as a transaction in a journal space. Commits are performed by applying the updates in the transaction and deleting the transaction from the journal. The transaction contains a list of block numbers to evict and the block data to write into the SAN storage. In contrast to the host-side journal approach we do not have access to the SSD data, therefore the eviction process has to start a transaction, write all the data to the journal on disk and then finish the transaction.

Staleness control. We modified this policy (both versions) to control staleness. The idea is to limit the size of the list of completed writes. By doing this we are effectively limiting the number of dirty blocks at any time, and therefore the staleness of the cache. By increasing this number we also increase performance as most writes can occur in place and overwrite the old blocks. By decreasing the limit we decrease performance and get closer to write-through: limit the list to size 0 is actually having the write-through policy.

6.4 Consistency analysis

We now explain how the two new write policies presented and how regular write-back and write-through handle consistency under certain conditions. We start by describing the variants analyzed:

- WT-S (Write-through safe): write to disk, SSD write, and then notify guest.
- WT (Write-through): write to disk, unbuffered SSD write, and then notify guest. This SSD write does not store the request in the internal SSD write buffer.
- WB (Write-back): regular write-back with SSD-to-disk block mapping persistent on SSD.

	Application inconsistency	Network File-system inconsistency	Staleness
Guest VM crash	~~WT-S~~ ~~WT~~ ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~
Hypervisor crash	~~WT-S~~ WT [1] ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [2] ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [4] WB-O [4] WB-JH [4] WB-JS [4]
SSD failure	~~WT-S~~ ~~WT~~ ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [2] ~~WB-O~~ WB-JH [3] ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [4] WB-O [4] WB-JH [4] WB-JS [4]
Host failure	~~WT-S~~ WT [1] ~~WB~~ ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [2] ~~WB-O~~ ~~WB-JH~~ ~~WB-JS~~	~~WT-S~~ ~~WT~~ WB [5] WB-O [5] WB-JH [5] WB-JS [5]

Table 6.2: Grid of resilience to failure and staleness for different write policies at different failure modes.

- WB-O (Ordered write-back): algorithm presented in section 6.3.1.
- WB-JH (Write-back with host-side journal): algorithm presented in section 6.3.2.
- WB-JS (Write-back with storage journal): algorithm presented in section 6.3.2.

We now analyze the consistency guarantees when reading the data from the SSD cache and from the SAN storage. Table 6.2 shows a grid of inconsistent and stale

states that write policies can get to for 4 failure modes. Guest and hypervisor crashes are recoverable after a reboot; we analyze whether there is inconsistencies after the recovery. SSD and host failures, on the other hand, are not recoverable making data in the SSD unavailable. We subdivide the possible inconsistencies in two. Application inconsistencies are the ones present when reading by the application and using the SSD cache. Network file system inconsistencies are the ones that appear when reading directly from the SAN storage, necessary in case the host-side SSD fails.

Each cell in table 6.2 lists the write-policies for which there is a possible inconsistency or stale state. Notice how all the write-back policies introduce staleness. Also notice how WB-O nor WB-JS do not lead to any possible inconsistency. We now explain these and some other inconsistencies in more detail. The subscript after the policy acronym is used as an index in the following list.

1. Hypervisor crashes and we wrote to the SAN storage but not to the SSD; write to the SSD was erroneously reported as OK. The SSD queue is limited and after is filled, there is queue build-up in RAM, which can be lost.
2. Evictions are not ordered, and anything can fail in the middle leading to possible inconsistencies.
3. The transaction (all evictions atomic) can fail in the middle and without the ability to restart the transaction we are inconsistent. Remember that writes in the middle of the commit do not have to be ordered. We need access to the SSD to restart the transaction.
4. Writes to the SAN storage are delayed, therefore reading from the SAN storage can return stale data.

6.5 Performance evaluation

We now present an evaluation of the new write policy presented in this chapter using an implementation in the Linux block layer. The questions we want to answer with this implementation are how much performance benefit do we get from it, and where does it come from. The ordered and journaled write-back at the host side were implemented. Journaled write-back at the storage server side was not implemented as it required complex modifications to the iSCSI protocol. Additionally, no implementation nor experiments were performed related to recovery after crashes.

6.5.1 Experimental Setup

We implemented our cache layer as a module for the Linux kernel 3.0.0. The cache block size is 4 KB and we use ARC as the replacement algorithm. The host was running at 2Ghz and configured to run with 512MB or 4GB of memory. A OCZ REVODRIVE PCI-express SSD was used as the host-side SSD attached to the host. The backing store was a RAID 5 using 7.2 RPM disks over iSCSI. The measured performance for these storage devices can be found in Table 6.2.

We evaluated the system with Postmark over EXT3 with a small amount of memory in order to maximize the amount of synchronous writes. Large memories were not tested, however, they may reduce the amount of synchronous writes and therefore reduce the potential performance benefits of these policies. EXT3 was configured to use ordered write-back journaling. Smaller memory sizes pressure memory pages to be evicted synchronously to disk. The workload was run directly over the host and without virtualization.

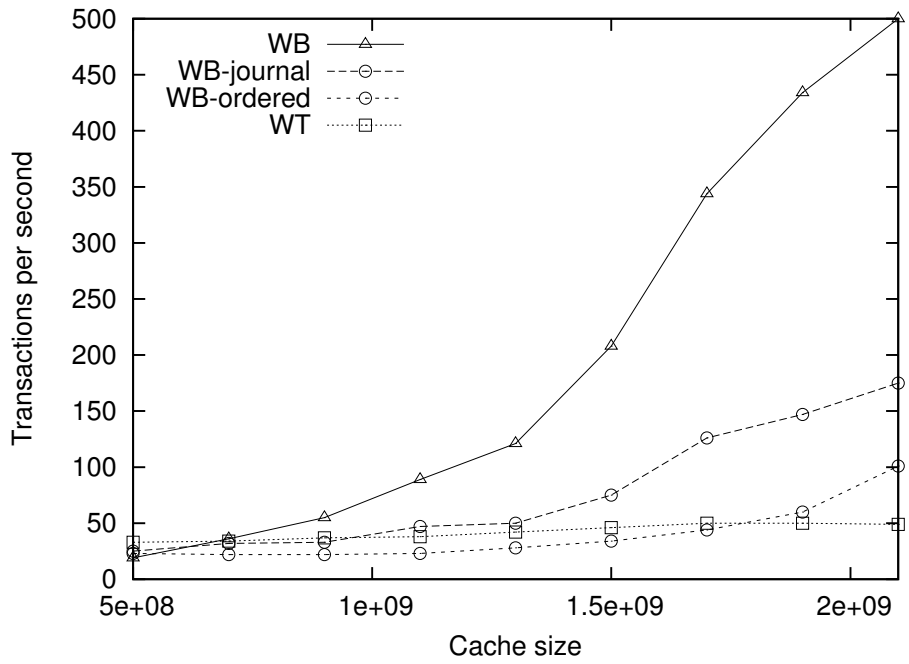

Figure 6.7: Number of transactions per second at different cache sizes for all four write policies.

6.5.2 Performance evaluation

In our first experiment we ran Postmark at different cache sizes for all 4 write policies: write-through-safe, write-back, write-back-ordered and journaled write-back at the host side. The journaled write-back was set up to limit the maximum dirty pages to 4096 at all times.

Figure 6.7 shows the number of transactions completed per second. As expected, write-through gets the lower performance for all cache sizes (except for ordered write-back which gets a slightly lower performance for small sizes), while write-back gets the best performance. The ordered write-back policy gets a slightly better performance, specially for large cache sizes. Journaled write-back on the other hand gets significantly more transactions than write-through, one third of write-back's performance.

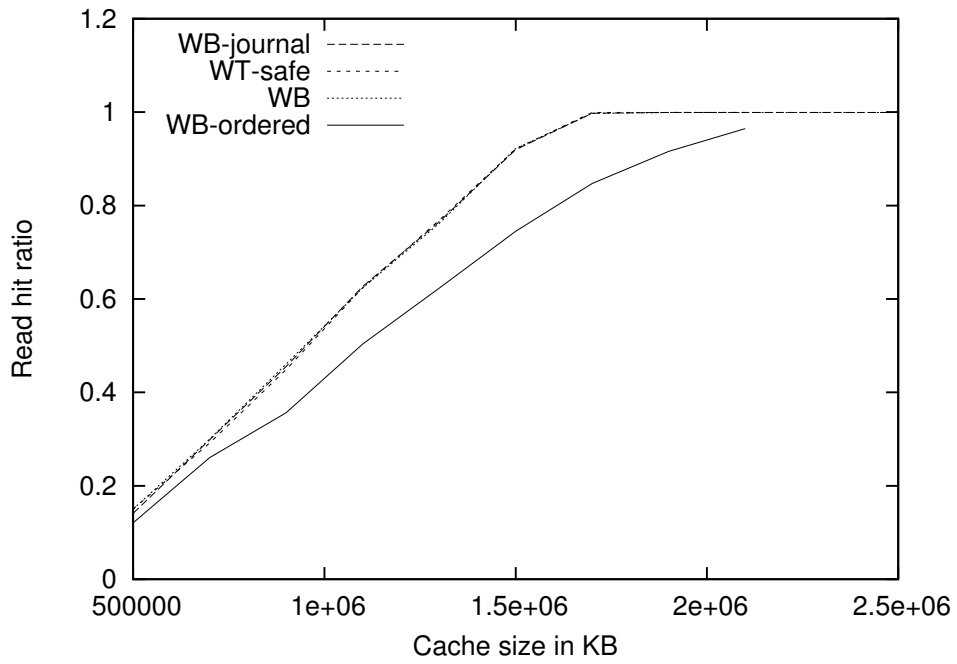

Figure 6.8: Hit ratio of Postmark reads as we increase the size of the cache for all four write policies.

We now explain where is the performance improvement coming from for write-back ordered when compared to the write-through policy. Ideally ordered write-back should perform as good as write-back, as there are no accesses to slow SAN storage in the original I/O path. However, by having to store copies of all writes to a block it wastes cache space and therefore reduce the hit rate of the application. Figure 6.8 shows this decrease in hit rate for ordered write-back to be 0.2 in average. Another reason for the poor performance of write-back ordered is that all writes require a new copy and therefore requires to evict something from the cache. Remember that an eviction of a dirty block means a read from SSD and then a write to disk. Additionally, write-back order needs to perform all the writes that occurred before the about-to-be evicted block. However, the average number of evicted dependencies is 2 for cache sizes below 900MB and 1 above 900MB.

Journaled write-back does not need to store copies of old blocks, and therefore

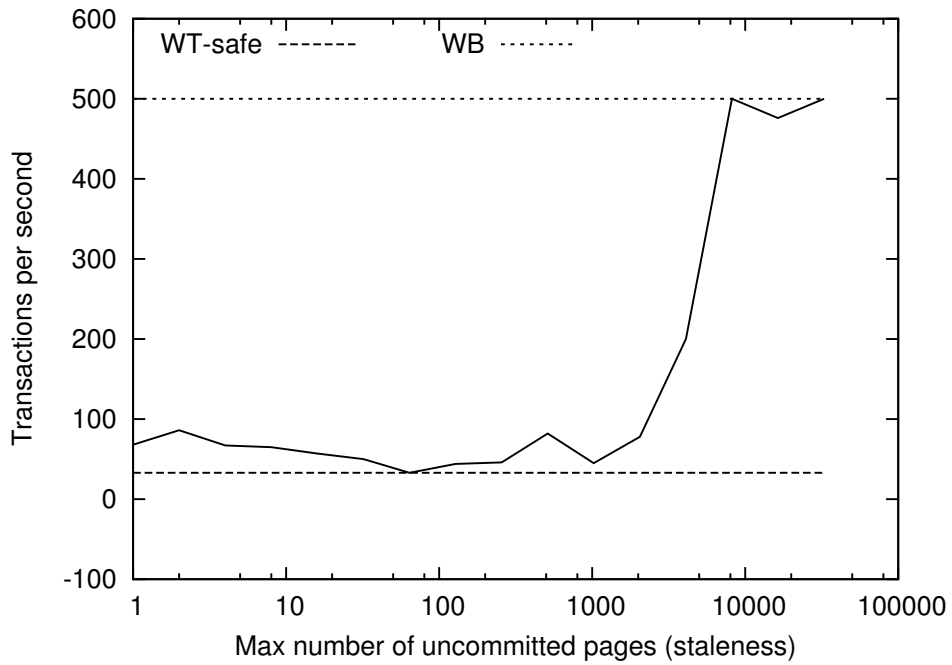

Figure 6.9: Postmark performance with 2.1GB of SSD cache for varying staleness set as the maximum number of dirty pages.

has minimal overhead relative to pure write-back. The reason why it performs worse than write-back in Figure 6.7 is that it limits the amount of staleness to 4096 dirty pages, thereby incurring additional evictions during transaction commits. To evaluate the sensitivity of ordered write-back performance to the allowable staleness of storage, we conducted an experiment where we fix the cache size to 2.1GB and use different maximum values for dirty pages, all for write-back journal. Figure 6.9 shows that we can tune journaled write-back to perform as write-through to write-back by varying this limit. This is an ideal knob to achieve an arbitrary application-defined performance/staleness trade-off.

6.6 Related work

Consistency for SSD caches has been addressed partially by Saxena et. al. in the Flashtier system [SSZ12]. The proposed cache provides persistence for the mappings

from the SSD address space to the disk and vice-versa. According to the authors, the persistence of this map is the only requirement for a consistent cache. They argue that providing ordering guarantees is a property of consistent storage devices and not of caches.

The Mercury SSD cache system [SBSMWS12] uses a write-through cache because, as indicated by the authors, consistency after crashes can only be achieved by writing synchronously to persistent storage before writes are reported to the applications. Additionally, they argue against the use of write-back caches because distributed hosts with local SSDs may have to implement complex and costly distributing agreement algorithms to synchronize between them. As part of their future work proposal, the authors mention that in case customers are willing to accept the loss of recently written data after a crash (staleness), then a way of providing crash consistency would be to order evictions with the original write-order by the application. In this thesis, we present a detailed design, implementation, and evaluation of an ordered write-back policy. Additionally, we propose and evaluate write-back policies that use host-side and storage side journaling to achieve further performance benefits while guaranteeing the same level of consistency and staleness guarantees as ordered write-back.

Other related ideas come from the field of asynchronous remote mirroring. For instance, Ji et. al. proposes [JVW03] delaying the writes to the secondary (mirror) site in order to improve performance, and sending the writes as single atomic updates with no risk of data loss or inconsistencies. In this thesis we design, implement and evaluate write-back caching policies that journal data writes which use an approach similar to those proposed in this earlier work in the different context of host-side SSD caching. In our work, we additionally address cache specific details such as page replacement and the fact that there is limited space in the cache for metadata.

6.7 Summary

The choice of write-policy for a cache can have a significant impact on performance and consistency. Current choices, write-through and write-back, are too limited in the possibilities; write-through is slow and safe, and write-back is fast and unsafe. We showed that the performance gains from write-back when compared to write-through are up to 5x, therefore blindly choosing write-through can be a waste of SSD performance. Based on this observation, we designed three write-policies, with the goal of being faster than write-through but providing more consistency guarantees than write-back. The consistency guarantees in write-back are achieved at the cost of some performance loss. Further, these policies are designed to allow trading off staleness for performance. The guarantees are provided by maintaining ordering in the evictions, and because evictions are the only way of updating the backend storage, this is enough to provide consistency.

Ordered write-back eliminates the chances of file system inconsistencies, but experiments showed that for Postmark, it can increase performance by 10 % for cache sizes close to the total working set size. Journaled write-back, on the other hand, showed a large performance increase when compared to write-through. Host-side journaling can introduce data inconsistency within the SAN storage if the host-side SSD (storing the journal) becomes inaccessible. When the journal is implemented in the server to support atomic committing of host-side journal transactions, there are no chances of inconsistencies.

Limitations. Our proposed write-back policies are limited to provide file system consistency through ordering. Another type of consistency is application consistency which can only be achieved through specific synchronization requests by the application. In order to support this type of consistency, the applications should be aware

of our cache layer and explicitly ask for synchronization between the cache and the backend storage.

CHAPTER 7

CONCLUSIONS

Consolidation trends are increasing the amount of data duplicated, and are introducing contention problems which lead to wasted cache space. We studied production traces at the storage level and benchmark traces at the memory level, and observed that cache duplication and contention are real and serious problems. We observed that for a server hosting two similar guests, more than of 40% of the pages in the buffer cache were duplicates. We also noticed that for 8 servers sharing the same cache, contention can lead to wasted space similar to the size of the working set of one of the workloads. Not only did we observe these problems on small consolidated scenarios (up to 8 shared workloads), but we also observed a linear increase in wasted space because of duplication and contention as the number of workloads increase.

We designed and implemented two systems to minimize duplication and contention. The one that deals with duplication was implemented at the buffer cache level and was able to boost applications' performance by 10% to 4x when compared to conventional location-addressed caches. To address cache contention, we implemented a system at the SSD cache level that partitions it per workload and was able to improve throughput by 17%. For caches that do not allow cache partitioning, such as CPU caches, we proposed a generalized model for Least Recently Used (LRU) caches capable of predicting wasted space for two or more applications using parameters like the rate at which blocks are reused. This work enables accurate cache provisioning for non-partitionable caches by accounting for wastage explicitly.

Besides duplication and contention, consolidation also creates a more subtle need: write-back policies with consistency guarantees. Consolidation has increased the performance requirements of storage systems, making the new high performance

flash-based devices ideal for caching. The issue is that these caches are placed below file systems which need ordering guarantees from the underlying storage. Unfortunately, write-through caches, the only option recommended and evaluated by researchers prior to our work, performs poorly for some workloads. We designed and implemented two write-policies based on write-back caches which provide good performance, controlled staleness and, more importantly, consistency guarantees. Experiments with micro-benchmarks showed that one of the proposed policies is capable of achieving performance similar to conventional write-back caches that do not provide any consistency guarantees.

The impact of these cache optimizations for applications and users is that they can get an increase in performance without augmenting the size of the caches. By removing duplicates or minimizing the negative effects of contention we can make space for other cached data which would otherwise miss the cache. A second nice property is that with these optimizations, the performance will always be equal or better than existing systems without these optimizations. Further, since running many workloads on the same machine is more the norm than an exception, we expect performance gains for the majority of systems using these cache optimizations.

Future work

We believe that an ideal solution to the problem of cache contention requires cooperation from hardware and storage vendors. CPU caches should be partitionable and the most straightforward way of doing it is through hardware. Storage appliances also suffer from the same problem as CPU caches, but for a different set of reasons. First, these appliances do not have sufficient knowledge to perform effective cache partitioning. They need techniques to identify and isolate individual workloads or machines. Second, they need efficient techniques to perform cache repartitioning when workload behavior changes due to the out-of-band I/O requirements involved.

The first future direction involves exploring new interfaces and protocols needed from hardware and storage, and the required operating systems changes to use them. As a second direction, we would like to have a deeper understanding of cache contention. For example, we would like to formally prove the existence of wastage for any cache policy and memory level and understand why it occurs.

BIBLIOGRAPHY

[AS95] Sedat Akyurek and Kenneth Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[Axb07] Jens Axboe. blktrace user guide, February 2007.

[BABD00] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *IEEE MICRO*, 2000.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN*, 2000.

[BDD+02] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Soft.*, 28(2):135–151, 2002.

[BEP11] Gianfranco Bilardi, Kattamuri Ekanadham, and Pratap Pattnaik. Efficient stack distance computation for priority replacement policies. In *ACM International Conference on Computing Frontiers*, 2011.

[BGU+09] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reorganization for self-optimizing storage systems. In *Proc. of USENIX FAST*, February 2009.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM76] A. P. Batson and A. W. Madison. Measurements of major locality phases in symbolic reference strings. In *Proc. of ACM Sigmetrics*, 1976.

[boo] Bootchart. http://www.bootchart.org.

[Car81] John L. Carr, Richard W. Hennessy. Wsclock-a simple and effective algorithm for virtual memory management. *SIGOPS Oper. Syst. Rev.*, 15:87–95, December 1981.

[CAVL09] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proc. of the USENIX Annual Technical Conference*, June 2009.

[CH81] R. Carr and J. Hennessy. WSCLOCK – A simple and efficient algorithm for virtual memory management. In *Proc. of ACM SOSP*, 1981.

[CO72] W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In *Proc. of AFIPS Conference*, 1972.

[Den68a] P. J. Denning. Thrashing: Its causes and prevention. In *Proc. of AFIPS Fall Joint Computer Conference*, 1968.

[Den68b] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[Den80] Peter J. Denning. Working sets past and present. In *IEEE Tran. on Software Engineerng*, 1980.

[Den06] Peter Denning. *The Locality Principle, Communication Networks And Computer Systems (Communications and Signal Processing)*. Imperial College Press, London, UK, UK, 2006.

[Dra] DragonFly BSD HammerFS. www.dragonflybsd.org/hammer/.

[DS02] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of ISCA*, 2002.

[EMC] EMC Corporation. EMC Invista. http://www.emc.com/products/software/invista/invista.jsp.

[fio] Fio. http://linux.die.net/man/1/fio.

[FY83] D. Ferrari and Y.-Y. Yih. VSWS: The variable-interval sampled working set policy. In *IEEE Trans. on Software Engineering*, 1983.

[GAW09] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, 2009.

[GAW11] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, New York, NY, USA, 2011. ACM.

[GF78] R. K. Gupta and M. A. Franklin. Working set and page fault frequency replacement algorithms: A performance comparison. In *IEEE Transactions on Computers*, 1978.

[Gil08] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *Proc. of the USENIX File and Storage Technologies*, Feburary 2008.

[GJMT03] Pawan Goyal, Divyesh Jadav, Dharmendra S. Modha, and Renu Tewari. Cachecow: Qos for storage system caches. In *Proceedings of the 11th international conference on Quality of service*, IWQoS'03, 2003.

[GKAK10] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. Basil: Automated io load balancing across storage devices. In *FAST*, pages 169–182, 2010.

[GLV+08] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. *Proc. of the USENIX OSDI*, December 2008.

[GMV10] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: handling throughput variability for hypervisor io scheduling. OSDI'10, 2010.

[GPG+11] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, 2011.

[GS00] Jim Gray and Prashant Shenoy. Rules of Thumb in Data Engineering. *Proc. of the IEEE International Conference on Data Engineering*, February 2000.

[GUB+08] Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami. The Case for Active Block Layer Extensions. *ACM Operating Systems Review*, 42(6), October 2008.

[Hag87] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. *Proc. 11th ACM Symposium on Operating System Principles*, November 1987.

[HHS05] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. In *Proc. of the ACM SOSP*, October 2005.

[IBM] IBM Corporation. IBM System Storage SAN Volume Controller. http://www-03.ibm.com/systems/storage/software/virtualization/svc/.

[Int09] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2009.

[iom] Iometer. http://www.iometer.org.

[isc04] Internet small computer systems interface (iscsi). *RFC 3720*, April 2004.

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[JCZ05] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proc. of the USENIX Annual Technical Conference*, April 2005.

[JDT05] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proc. of the USENIX Conference on File And Storage Systems*, 2005.

[JNW08] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[JVW03] Minwen Ji, Alistair Veitch, and John Wilkes. Seneca: Remote mirroring done write, 2003.

[KCK+] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-

performance unified buffer management scheme that exploits sequential and looping references. OSDI'00.

[KCK+00] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. of USENIX OSDI*, 2000.

[KDLT04] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. *Proc. of the USENIX Annual Technical Conference*, 2004.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 1983.

[KHW91] Y. Kim, M. Hill, and D. Wood. Implementing stack simulation for highlyassociative memories. In *Proc. of ACM SIGMETRICS*, 1991.

[KR10] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[KRDZ10] Sajib Kundu, Raju Rangaswami, Kaushik Dutta, and Ming Zhao. Application performance modeling in a virtualized environment. In *Proc. of the IEEE HPCA*, January 2010.

[krm08] taeho kgil, david roberts, and trevor mudge. improving nand flash based disk caches. In *proc. of isca*, 2008.

[KS00] M. Karlsson and P. Stenstrom. An analytical model of the working-set sizes in decision-support systems. In *Proc. of ACM SIGMETRICS*, 2000.

[KVR10] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Generalized erss tree model: Revisiting working sets. *Perform. Eval.*, 2010.

[KVR11] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Estimating application cache requirement for provisioning caches in virtualized systems. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:55–62, 2011.

[LAS+05] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proc. of the USENIX File and Storage Technologies*, 2005.

[LEB+09] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. of the USENIX File and Storage Technologies*, February 2009.

[LLD+08] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of IEEE HPCA*, 2008.

[LSB08] The Linux Standard Base. *http://www.linuxbase.org*, 2008.

[McD] Richard McDougall. Filebench:application level file system benchmark.

[MCGC11] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.

[McK04] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[MG99] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: a technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[MGST70a] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[MGST70b] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 1970.

[MIG03] Charles B. Morrey III and Dirk Grunwald. Peabody: The Time Travelling Disk. In *Proc. of the IEEE/NASA MSST*, 2003.

[MM04] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 2004.

[MMHF09] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened Page Sharing. In *Proc. of the Usenix Annual Technical Conference*, June 2009.

[NAS] NAS Parallel Benchmarks (NPB). http://www.nas.nasa.gov/resources/software/npb.html.

[Net] Network Appliance, Inc. NetApp V-Series of Heterogeneous Storage Environments. http://media.netapp.com/documents/v-series.pdf.

[New] New Implementation of a Log-structured File System (NILFS). www.nilfs.org.

[Ope] Open Source. Dyninst: An Application Program Interface (API) for Runtime Code Generation. Online, http://www.dyninst.org/.

[OS93] Cyril U. Orji and Jon A. Solworth. Doubly distorted mirrors. In *Proceedings of the ACM SIGMOD*, 1993.

[PGG+95] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of ACM SOSP*, 1995.

[PGSK09] C.M. Patrick, R. Garg, S.W. Son, and M. Kandemir. Improving i/o performance using soft-qos-based dynamic storage cache partitioning. In *CLUSTER '09*, 2009.

[PSPK09] Ramya Prabhakar, Shekhar Srikantaiah, Christina Patrick, and Mahmut Kandemir. Dynamic storage cache allocation in multi-server architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[QD02] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. *Proc. of the USENIX File And Storage Technologies*, January 2002.

[QP06a] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006.

[QP06b] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO2009*, 2006.

[RCP08] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, Inexpensive Content-Addressed Storage in Foundation. *Proc. of USENIX Annual Technical Conference*, June 2008.

[RD] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. SIGMETRICS '90.

[RLLS] R. Rajkumar, Chen Lee, J.P. Lehoczky, and D.P. Siewiorek. Practical solutions for qos-based resource allocation problems. In *Real-Time Systems Symposium, 1998.*

[RO91] M. Rosenblum and J. Ousterhout. The Design And Implementation of a Log-Structured File System. *Proc. 13th Symposium on Operating System Principles*, October 1991.

[RSG93] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proc. of ISCA*, 1993.

[RT03] Ohad Rodeh and Avi Teperman. zfs " a scalable distributed file system using object disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, MSS '03, pages 207–, Washington, DC, USA, 2003. IEEE Computer Society.

[SBSMWS12] Luis Pabn Steve Byan, James Lentini, Christopher Small, and Inc. Mark W. Storer, NetApp. Mercury: host-side flash caching for the datacenter. 2012.

[Sit96] Richard L. Sites. It's the memory, stupid! *Microprocessor Report*, 1996.

[SLG+09] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proccedings of the 7th conference on File and storage technologies*, 2009.

[SRD04] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 2004.

[SSZ12] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier a lightweight, consistent and durable storage cache. In *EuroSys*, pages 267–280, 2012.

[STS08] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *IEEE MICRO*, 2008.

[STW92] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 1992.

[SVS12] Priya Sehgal, Kaladhar Voruganti, and Rajesh Sundaram. SLO-aware Hybrid Store. *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2012.

[TASS09] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *proc. of asplos*, 2009.

[Tra] Transaction Processing Performance Council (TPC). TPC Benchmarks. *http://www.tpc.org/information/benchmarks.asp*.

[TSW92a] D. Thiebaut, H.S. Stone, and J.L. Wolf. Improving disk cache hit-ratios through cache partitioning. *Computers, IEEE Transactions on*, 41(6):665 –676, jun 1992.

[TSW92b] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Trans. Comput.*, 1992.

[Twe98] S. C. Tweedie. Journaling the Linux ext2fs File System. *The Fourth Annual Linux Expo*, May 1998.

[UGB+08] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. *Proc. IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, February 2008.

[VAN08a] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Proc. of ACM Middleware*, 2008.

[VAN08b] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proc. of ACM ICS*, 2008.

[Wal02a] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *Proc. of USENIX OSDI*, 2002.

[Wal02b] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[WOT+96] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *Proc. of ISCA*, 1996.

[WW02a] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference*, 2002.

[WW02b] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of the USENIX File And Storage Technologies*, February 2008.

[ZPS+04a] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of ASPLOS*, 2004.

[ZPS+04b] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page

miss ratio curve for memory management. *SIGOPS Oper. Syst. Rev.*, 2004.

[ZYKW02] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *Proc. of USENIX File and Storage Technologies*, January 2002.

VITA

RICARDO KOLLER

November 6, 1982	Born, Cochabamba, Bolivia
2007–2012	Research Assistant Florida International University Miami, Florida
2005	B.S., Systems Engineering Bolivian Catholic University New York, New York

PUBLICATIONS AND PRESENTATIONS

Luis Useche, Ricardo Koller, Raju Rangaswami, Akshat Verma, (2011). *Truly Non-Blocking Writes*. Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems.

Ricardo Koller, Akshat Verma, and Raju Rangaswami, (2011). *Estimating Application Cache Requirement for Provisioning Caches in Virtualized Systems*. Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.

Akshat Verma, Gautam Kumar, Ricardo Koller, Aritra Sen, (2011). *CosMig: Modeling the Impact of Reconfiguration in a Cloud*. Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.

Ricardo Koller, Akshat Verma, and Raju Rangaswami, (2010). *Generalized ERSS Tree Model: Revisiting Working Sets*. Proceedings of the IFIP International Symposium on Computer Performance, Modeling, Measurements and Evaluation.

Ricardo Koller, Akshat Verma, Anindya Neogi, (2010). *WattApp: An Application Aware Power Meter for Shared Data Centers*. Proceedings of the IEEE International Conference on Autonomic Computing.

Ricardo Koller and Raju Rangaswami, (2010). *I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance*. Proceedings of the USENIX Conference on File and Storage Technologies.

Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami, (2010). *SRCMap: Energy Proportional Storage Using Dynamic Consolidation*. Proceedings of the USENIX Conference on File and Storage Technologies.

Ricardo Koller, Raju Rangaswami, Joseph Marrero, Igor Hernandez, Geoffrey Smith, Mandy Barsilai, Silviu Necula, S. Masoud Sadjadi, Tao Li, and Krista Merrill, (2008). *Anatomy of a Real-time Intrusion Prevention System*. Proceedings of the IEEE International Conference on Autonomic Computing.

Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami, (2008). *The Case for Active Block Layer Extensions*. Proceedings of IEEE International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability.

HONORS AND AWARDS

2011	Overall outstanding graduate student award Florida International University Miami, Florida
2010	International research fellowship IBM/PIRE Miami, Florida