6-20-2016

Optimizing Main Memory Usage in Modern Computing Systems to Improve Overall System Performance

Daniel Jose Campello
Florida International University, dcamp020@fiu.edu

DOI: 10.25148/etd.FIDC000755

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

OPTIMIZING MAIN MEMORY USAGE IN MODERN COMPUTING SYSTEMS TO IMPROVE OVERALL SYSTEM PERFORMANCE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Daniel Campello

2016

To: Interim Dean Ranu Jung
College of Engineering and Computing

This dissertation, written by Daniel Campello, and entitled Optimizing Main Memory Usage in Modern Computing Systems to Improve Overall System Performance, 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

Jason Liu

Gang Quan

Ming Zhao

Raju Rangaswami, Major Professor

Date of Defense: June 20, 2016

The dissertation of Daniel Campello is approved.

Interim Dean Ranu Jung
College of Engineering and Computing

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

Florida International University, 2016

DEDICATION

To Isaac, Tabata, Marina and Juan José.

ACKNOWLEDGMENTS

I wish to acknowledge my research sponsors including NSF via CNS-1320426, CNS-1018262, CNS-0747038, gifts from Intel and NetApp, and USENIX's travel scholarships.

ABSTRACT OF THE DISSERTATION

OPTIMIZING MAIN MEMORY USAGE IN MODERN COMPUTING SYSTEMS TO IMPROVE OVERALL SYSTEM PERFORMANCE

by

Daniel Campello

Florida International University, 2016

Miami, Florida

Professor Raju Rangaswami, Major Professor

Operating Systems use fast, CPU-addressable main memory to maintain an application's temporary data as anonymous data and to cache copies of persistent data stored in slower block-based storage devices. However, the use of this faster memory comes at a high cost. Therefore, several techniques have been implemented to use main memory more efficiently in the literature. In this thesis we introduce three distinct approaches to improve overall system performance by optimizing main memory usage.

First, DRAM and host-side caching of file system data are used for speeding up virtual machine performance in today's virtualized data centers. The clustering of VM images that share identical pages, coupled with data deduplication, has the potential to optimize main memory usage, since it provides more opportunity for sharing resources across processes and across different VMs. In our first approach, we study the use of content and semantic similarity metrics and a new algorithm to cluster VM images and place them in hosts where through deduplication we improve main memory usage.

Second, while careful VM placement can improve memory usage by eliminating duplicate data, caches in current systems employ complex machinery to manage the cached data. Writing data to a page not present in the file system page cache causes the operating system to synchronously fetch the page into memory, blocking the writing process. In this thesis, we address this limitation with a new approach to managing page writes involving

buffering the written data elsewhere in memory and unblocking the writing process immediately. This buffering allows the system to service file writes faster and with less memory resources.

In our last approach, we investigate the use of emerging byte-addressable persistent memory technology to extend main memory as a less costly alternative to exclusively using expensive DRAM. We motivate and build a tiered memory system wherein persistent memory and DRAM co-exist and provide improved application performance at lower cost and power consumption with the goal of placing the right data in the right memory tier at the right time. The proposed approach seamlessly performs page migration across memory tiers as access patterns change and/or to handle tier memory pressure.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Computer systems use memory to store both instructions and data required by applications and the operating system (OS). There are several types of memory (registers, SRAM, DRAM, Disks, Flash, etc.) that service operations at different speeds; the faster a memory is, the more expensive it is. To make better use of the capabilities of faster memory but incur the cost of the slower memory, systems employ different techniques to mix the different types of memory. One of such technique is caching. A cache is a smaller, faster memory which stores copies of the data from recently (or frequently) used slower memory locations. Another technique is tiering, whereas the different types of memory with similar capabilities (i.e., byte-addressability) are organized as different tiers in a multi-tier memory system. For all of these techniques one of the most important metrics to evaluate the performance of a system is how many accesses are serviced from the faster memory. A higher access rate to faster memory leads to lower latency for a greater percentage of all data accesses, and as a consequence, better application performance.

In today's virtualized data centers, DRAM and host-side caching of file system data are used for speeding up VM performance [BLM+12, KMR+13, KMR15]. Big cloud data centers make use of virtualization techniques to deploy thousands of virtual machines in hundreds of hosts. However, their growing popularity has led to the problem of virtual machine sprawl, which translates into extremely large working set sizes for the host machines.

When trying to optimize main memory usage, one of the biggest problems to deal with is the workload's working set size. The bigger the working set, the harder it is to fit the required data into main memory and the more likely it is for cache misses (the requested data is not present in the cache, requiring to access slower memory) to happen. Thus, the use of deduplication has become popular nowadays. This technique removes duplicates of pages across guests virtual machines from the host machine main mem-

ory [AEW09, MFG+12, SK12]. The elimination of duplicates effectively increases the amount of available space in main memory to be used by VMs to service application allocations or to cache even more file system data. Additionally, all duplicates are transformed into a single version of the data that becomes then shared across different VMs improving the actual usage of that portion of main memory.

The clustering of VM images that share identical pages, coupled with data deduplication, has the potential to optimize memory usage even more, given the increased opportunity for removal of duplicate pages. In this thesis, we study the use of content and semantic similarity metrics and a new clustering algorithm to cluster virtual machines images and place them in hosts so as to improve data deduplication and overall memory usage:

> *We develop* **CORIOLIS***, **a scalable system that*** **analyzes** ***virtual machine images and automatically clusters them based on content and/or semantic similarity**. We show that similarity analysis can help with virtual machine placement in hosts to improve memory deduplication ratio and, as a consequence, cache efficiency. Images with high semantic similarity are likely to exhibit higher number of duplicate pages in main memory and/or host-side caches, given that they are likely to execute same software (i.e., load same libraries and executables). Similarly, images with higher content similarity can benefit more from deduplication performed at a shared management server (e.g., vSphere). Additionally, we show that similarity analysis can also help in the* planning *of many management activities (e.g., migration, system administration). However, clustering images based on their similarity – content or semantic – requires large scale data processing and does not scale well.* CORIOLIS *uses (i) asymmetric similarity semantics and (ii) a hierarchical clustering approach with a data access requirement that is linear in the number of images.*

While careful VM placement can help optimize main memory usage by eliminating duplicate data, caches in current systems employ complex machinery to manage the cached

data. Significantly, the type of accesses issued by a workload and how each access type is handled within the caching layer influences the memory usage and performance achieved by systems.

File systems are used to store and retrieve data in the form of files, typically from slow, block-based, persistent storage (i.e., hard disk drives, solid-state drives). Caching and buffering file data within the operating system main memory is a key performance optimization that has been prevalent for over four decades [DN65, RT74]. For real-world workloads, most file reads can be made to exhibit main memory accesses latency by *pre-fetching* (i.e., pre-populating the file system cache). File writes, however, must be written to persistent storage at some point in time for durability, which incurs into block-based storage high latency.

The OS caches file data in units of pages, seamlessly fetching pages into main memory from the backing store, when necessary, as they are read or written to by a process. For file writes, the OS allows two different mechanisms that take advantage of caching: *(i)* asynchronous, wherein the OS uses write-back caching of the written data, or *(ii)* synchronous, wherein the OS uses write-through caching. Asynchronous file writes exhibit main memory's low latency if the access involves a *hit* in the OS' page cache. However, synchronous writes always exhibit high latency to write to slow persistent storage; they are useful for applications that need to ensure the durability of their data.

In this thesis, we improve system performance with the next approach which make a more efficient use of memory to make asynchronous file writes fast, regardless of whether their accesses incur in a *cache hit* or *cache miss*:

> *Writing data to a page not present in the file-system page cache causes the operating system to synchronously fetch the page into memory, always blocking the writing process.* ***Non-blocking writes eliminate such blocking by buffering the written data elsewhere in memory and unblocking the writing process immediately****. Subsequent*

reads *to the updated page locations are also made non-blocking. This new handling of writes to non-cached pages allow processes to overlap more computation with I/O and improves page fetch I/O throughput by increasing fetch parallelism.*

Main memory is not exclusively used to cache file system data residing in slow block-based storage but it is also used to maintain anonymous memory page containing applications temporary data. While improving caching effectiveness can improve overall system performance for a limited amount of main memory it cannot help to host applications with greater requirements for anonymous memory than the available system DRAM. Paging is a technique that has been used to extend the amount of fast main memory with the use of slow block-based storage for cold data. While paging can help augment the amount of available memory for applications to use, in many cases, the performance hit involved with paging is limiting in its adoption by certain memory intensive workloads. Emerging byte-addressable persistent memory opens up a new design alternative where slower memory can be used as an additional tier of main memory and in this way augment the total amount of memory available for applications to store their temporary data. While using different types of byte-addressable memory as separate tiers in a multi-tier system is a compelling solution to host increasingly big data sets, it presents its own challenges. In this thesis, we present solutions that improve an application's capabilities of allocating big data sets in main memory by using persistent memory as a new tier in a multi-tier memory system. Our solutions implement dynamic migration mechanisms to place the data in the right tier according to the frequency of accesses.

Applications rely on main memory to store their data while it is being processed by the CPU. Today, DRAM is used as main memory due to its access latency, bandwidth, and cost relative to other CPU-addressable memory technologies. However, systems today are limited in their use of DRAM due to limitations posed by capacity, cost, and power consumption. Byte-addressable persistent memory devices offer a unique set of prop-

erties that make them well-suited as main memory extension devices to address these limitations. We propose **MULTI-CLOCK*, a tiered memory system wherein persistent memory and DRAM co-exist and provide improved application performance at lower cost and power consumption with the goal of placing the right data in the right memory tier and at the right time***. MULTI-CLOCK *builds upon the well-understood* CLOCK *page replacement algorithm to seamlessly handle migration across memory tiers as access patterns change and/or to handle tier memory pressure. It is entirely transparent to, and backward compatible with, existing applications. We discuss the motivation for* MULTI-CLOCK*, outline a system design, and implement and evaluate a* MULTI-CLOCK *prototype in Linux. Our micro-benchmark based evaluation demonstrates that our implementation of* MULTI-CLOCK *exhibits fairly low migration overhead. Also, while static tiering is unable to do so,* MULTI-CLOCK *implements dynamic fairness of resource allocation between processes.*

We have organized the rest of this thesis in the following way. In chapter 2, we present our problem statement and we elaborate in its description and significance. In chapter 3, we discuss some of the background knowledge related to the different areas in which this thesis is focused. In chapters 4, 5 and 6, we present our design, implementation and evaluation of CORIOLIS, *Non-blocking Writes*, and MULTI-CLOCK, respectively. We conclude by discussing the body of related work in chapter 7.

CHAPTER 2

PROBLEM STATEMENT

This chapter introduces the core research problem addressed in this proposal. First, we present a statement of the thesis, and then we elaborate on its significance, introduce specific challenges that we address, and our contributions.

2.1 Thesis Statement

In this thesis, we improve overall system performance by:

(i) creating new clustering algorithms that aid in virtual machine placement to take advantage of deduplication mechanisms at the hypervisor level and, as a consequence, optimize memory usage in virtualized environments,

(ii) eliminating process blocking and reducing memory resources required to service file writes to pages that are not present in the OS cache in a way that is transparent to applications, and

(iii) designing and implementing new memory tiering solutions that integrate emerging byte-addressable persistent memory into main-memory and dynamically migrate pages into appropriate tiers depending on page importance.

2.2 Thesis Statement Description

The first contribution of this thesis is creating new clustering algorithms to aid in virtual machine placement to take advantage of deduplication mechanisms at the hypervisor level and, as a consequence, optimize memory usage in virtualized environments. By carefully collocating similar virtual machines in hosts of a data center, hypervisor cache efficiency and memory usage can be improved. Higher cache efficiency, involves handling less misses

(and as a consequence, less evictions) and higher hit rates which in turn lead to reduced latency experienced by VM-hosted applications. Higher memory usage, on the other hand, involves higher degree on memory pages sharing between processes across different VMs, reducing the total memory footprint of workloads.

The second contribution of this thesis is the elimination of process blocking and reduction of memory resource required to service file writes to pages that are not present in the cache in a transparent way to applications. General purpose operating systems today implement blocking asynchronous writes on page cache misses. This thesis proposes new solutions that eliminate blocking on asynchronous writes within file systems. Certain configurations of these solutions are able to process writes requiring less memory resources by avoiding complete-page allocations for sub-page size writes, and in this way to use the cache more efficiently. The impact of these solutions spawns a vast number of existing computer systems, ranging from mobile computing to the big data centers.

The final contribution of this thesis is designing and implementing new memory tiering solutions that integrate emerging byte-addressable persistent memory into main-memory and dynamically migrate pages into appropriate tiers depending on page access frequency. This thesis proposes the use of persistent memory as an additional tier of main memory augmenting the total amount of memory available for applications without incurring the high cost of DRAM. We solve the challenges related to a multi-tier memory system by enabling the operating system to dynamically and transparently migrate pages between tiers according to their use by applications.

2.3 Thesis Impact

Our first contribution of this thesis has the impact of improve main memory usage in virtualized data centers. This, leads to reduced latency experienced by VM-hosted applications

which ultimately means an increase in the quality of service experienced by virtualization clients.

Our second and third contribution of this thesis, aim to reduce the latency experienced by accessing slow memory or secondary storage. Mobile systems are one area of big interest. Given the interactive nature of these systems, any increased responsiveness from applications due to reduced blocking on file operations is well noticed by users. Our second contribution has the potential of increasing the parallelism of I/O operations. This, in turn, improves throughput since storage devices offer greater performance at higher levels of I/O parallelism, making the whole system better performing.

Our third contribution addresses the issue of the existing demand for high capacity, low cost, and low power consumption memory while retaining the existing performance levels. New generation of big data analytics applications in the enterprise and cloud, being inherently memory intensive workloads, are example applications with these demands. Our contribution on mobile systems, solves the problem with the existing cost and power constraints that limit DRAM size and, in consequence, performance in today's system.

Finally, all of our contributions have the benefit of reducing network traffic when centralized storage is used as secondary storage. This characteristic becomes very relevant for big data centers where the network bandwidth can become a scarce and costly resource. Any additionally network bandwidth that becomes available can, in turn, allow for higher degrees of virtual machine consolidation in virtualized environments.

CHAPTER 3

BACKGROUND

This chapter introduces concepts and background knowledge used in the remaining chapters of this proposal. We start by introducing how deduplication techniques are used in current system to eliminate duplicated data and how this can improve cache hit rates and memory usage. We then continue by introducing how applications use existing operating system mechanisms to take advantage of the in-memory cache of file system data. We conclude by presenting an overview of the characteristics of emerging memory technologies and motivate their use as main memory extension.

3.1 Cache Deduplication

Deduplication is a compression technique used to reduce the size of the input data by eliminating duplicate copies of the same contents. Data is processed in chunks of fixed or variable size and all unique chunks are identified and stored, while chunks with repeated data are discarded and a reference to the existing chunk with the same data is stored in place.

Deduplication techniques differ from other compression techniques such as dictionary based compression algorithms (i.e., LZ algorithms). Whereas these algorithms aim to compress a limited amount of data, typically within a single file, deduplication is applied to big sources of data such as block devices containing a file system instances or a set of partitions.

Deduplication has been previously used to remove duplicates of pages across guests virtual machines from the host machine main memory [AEW09, MFG+12, SK12]. The elimination of duplicates effectively increases the amount of cache space available for caching additional important data.

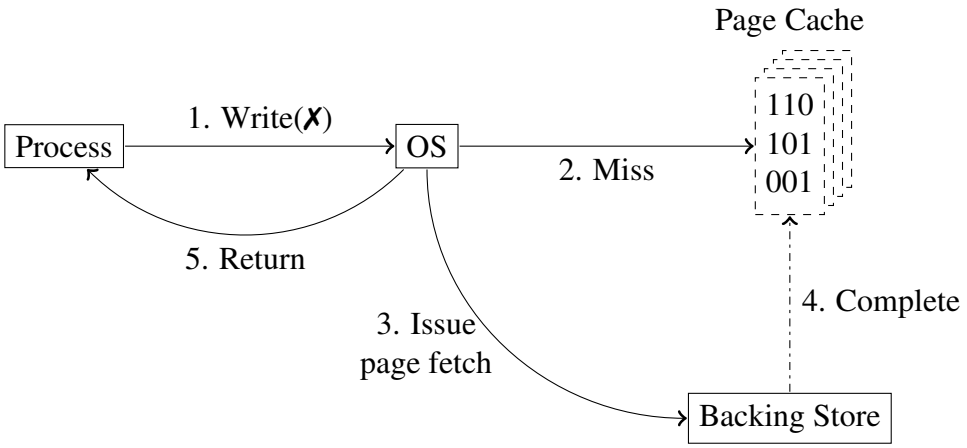


Figure 3.1: **Anatomy of a write.** The first step, a write reference, fails because the page is not in memory. The process resumes execution (Step 5) only after the blocking I/O operation is completed (Step 4). The dash-dotted arrow represents a slow transition.

3.2 Operating System Caching

Applications can access file data in two ways: *unsupervised*, by memory mapping a portion of the file to the address-space, and *supervised*, using the operating system (OS) file system call interface. For file writes, the OS allows three different mechanisms for supervised accesses: *(i)* asynchronous, wherein the OS uses write-back caching of the written data within its page cache, *(ii)* synchronous, wherein the OS uses write-through caching, or *(iii)* O_DIRECT, wherein the write is not cached in the page cache at all. The former two are dominant write modes for file system oriented accesses today. Asynchronous file writes exhibit memory's low latencies if the access involves a *hit* in the OS' page cache. However, synchronous writes always exhibit high latencies to write to slow, block-based, persistent storage. Applications rely on synchronous writes to ensure the durability of their data.

Besides handling application durability requests, modern operating systems control data staleness by forcing the durability of dirty data in the OS page cache via mechanisms such as Linux's flusher threads [Lov10].

While servicing writes, page fetch behavior in file systems is caused because of the mismatch in data access granularities: *bytes* accessed by the application, and *pages* ac-

Parameter	DDR-DRAM	PM
Capacity per CPU	100s of GBs	Terabytes
Read Latency	1x	2x to 4x
Write Bandwidth	1x	1/8x to 1/4x
Estimated Cost	5x	1x
Endurance	10^{16}	10^{6} to 10^{8}

Table 3.1: **Comparison of Memory Technologies [DRZ+16].** Persistent memory characteristics were derived from PCM and ReRAM technologies [QSR09, Cro13].

cessed from storage by the operating system. To handle write references, the target page is synchronously fetched before the write is applied, leading to a fetch-before-write requirement [MBKQ96, Tan07]. This is illustrated in Figure 3.1. This *blocking* behavior affects performance since it requires fetching data from devices much slower than main memory. Today, main memory accesses can be performed in a couple of nanoseconds whereas accesses to flash drives and hard drives can take hundreds of microseconds to a few milliseconds respectively. We confirmed the page *fetch-before-write* behavior for the latest open-source kernel versions of BSD (all variants), Linux, Minix, OpenSolaris, and Xen.

3.3 Emerging Memory Technologies

Emerging byte-addressable persistent memory technology such as ReRAM, STT-RAM and PCM offer an attractive set of properties that are well-suited for extending main memory in response to the higher demand for memory capacity (Table 6). These new memories promise to offer byte-addressable memory access at a fraction of DRAM cost. When used to extend main memory their capability for persistence is irrelevant and can be used as volatile memory. Moreover, they can achieve latency and bandwidth properties that are within an order of magnitude of DRAM. Finally, they do not require power to retain data, greatly reducing the power consumption costs within a system which a system with large quantities of DRAM would incur.

DRAM based memory systems have two significant drawbacks pertaining to cost and power consumption. These drawbacks impact their usage in both enterprise servers and mobile systems. Today's multi- and many-core platforms support higher levels of workload parallelism (in mobile and server workloads) and multi-tenancy (in server workloads), placing greater demands on the memory system. Furthermore, the new generation of big data analytics applications in the enterprise and cloud are, inherently, memory intensive [spa, sap, Fit04, Sal]. Their workloads demand access to high-performance, yet low-cost, memory devices in preference to the much slower storage system. The use of persistent memory is rather appealing given the high cost and power demands imposed by DRAM technology.

CHAPTER 4

CORIOLIS: SCALABLE VM CLUSTERING FOR CACHE DEDUPLICATION

In this chapter we examine the virtualized environment of big data centers and we present a new clustering technique for virtual machine images, which when used to assign virtual machines to hosts, increases the opportunities of deduplication of identical pages, and as a consequence, optimizes memory usage through sharing.

4.1 VM Clustering: An Overview

Cloud computing lends a fundamental shift to how businesses view IT, from being capital-intensive to being a commodity that can be acquired on-demand and paid for as per usage. However, the growing popularity of cloud data centers has led to the problem of virtual machine sprawl. Standardization is a key principle that allows cloud providers to provide services on-demand and at a lower cost than what individual IT departments can do. System management costs reduce with standardization of software at all levels: operating systems, middleware, applications, and management tools [GP11, VVK+12].

We conjecture that classifying diverse virtualized servers in a cloud into clusters of similar virtual machines (VMs) can improve host-side cache efficiency and improve overall performance. We classify VM similarity into two types – content similarity and semantic similarity. Content similarity refers to data similarity in the raw files that constitute virtual machines. Semantic similarity refers to the similarity in the operating system, middleware, and application software present in two virtual machines.

Deduplication across VMs is a popular technique to improve resource efficiency. Examples include memory sharing between VMs on a host [Wal02] or deduplication during backup of virtual machines [VMW12]. Clustering based on semantic similarity can help place VMs with a similar application stack on a host and lead to greater memory sharing

across VMs. Clustering based on content similarity can help allocate similar VMs to one backup cluster and achieve better storage resource utilization within each cluster.

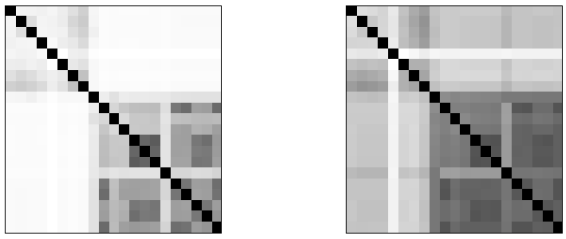
We develop and evaluate CORIOLIS, a framework for clustering images based on any given notion of similarity. Conventional clustering techniques require at least quadratic data access or worse, which is prohibitive for cloud environments with a large number of VMs. Further, clustering images based on the conventional symmetric notion of similarity leads to a uniform data access pattern; consequently, caching techniques that leverage popularity or locality for optimizing index lookup in deduplication systems [ZLP08, DSL10] are not applicable. CORIOLIS employs a novel tree-based VM clustering algorithm that consumes time that is only linear in the number of images. The algorithm uses an asymmetric notion of similarity to avoid computing all-pairs similarity values and a hierarchical order to introduce popularity in data access.

4.2 VM Similarity: Types and Applications

The similarity across VMs in enterprise data centers and clouds has been studied extensively in the context of data deduplication [CAVL09, DSL10, JM09, KR10, ZLP08]. In this section, we discuss both content and semantic similarity and then discuss how such similarity can be utilized for streamlining system management tasks.

4.2.1 Content Similarity

The classical notion of similarity is that of *content*, whereby a subset of the bytes contained within the images are identical. Identical content can occur either in the form of whole or partial files [MB11] and techniques to detect similar content have ranged from whole file and fixed size chunking to more sophisticated variable size chunking [JM09, ZLP08]. Content similarity across VM images occurs because of the use of similar operating system



(a) Content similarity (b) Semantic similarity

Figure 4.1: **Distribution of content and semantic similarity for 25 VM image pairs.** Image pairs obtained from two production data centers. Small block (i, j) denotes similarity between images i and j. White regions denoting less than 10% similarity while black regions denoting more than 90% similarity. The heatmap itself is symmetric across the diagonal; pairwise similarity is a symmetric function.

instances as well as the widespread use of master images within publicly available virtual appliance libraries to create VMs.

In both cases, the resulting VM images are subsequently modified from the original to varying extents owing to configuration changes, software updates, and user-specific application data. Content similarity is useful in minimizing the amount of data that needs to be managed for a task involving a collection of VMs (e.g., VM backup [VMW12] or Virtual Image Library [ABM+11]). A recent large-scale study of VM images in a production IaaS cloud investigates such content similarity [JPZ+11]. This study found that the distribution of *content similarity across images is skewed* and that individual VM images tend to be similar to a small subset of images than to the entire image population leading to clusters of similar images. They also noted that *computing pair-wise similarity is very expensive* and reported results for only 30% of their image collection due to scalability issues.

We performed a similarity study across VM images from 2 production data centers. The first set of 9 images is from a large-scale enterprise data center at IBM. The latter set of 12 images is from the Computer Science department's small-scale data center at Florida International University. The former set of images is relatively more diverse reflecting the

needs typical of a large-scale enterprise data center. The latter set of images are more homogeneous reflecting limited software needs. Figure 4.1(a) depicts pairwise content similarity distribution across the image sets. Content similarity is computed as ratio of the number of shared bytes in the two images to the sum of all unique bytes across the images. Rows and columns are ordered using all images belonging to the former image set first followed by those in the latter image set. Darker shades denote higher levels of similarity. We can see the distinct formation of two high similarity clusters, wherein images within the cluster are more similar to each other than images outside of the cluster. Such high amount of content similarity within clusters of images motivates content-based optimizations.

4.2.2 Semantic Similarity

Semantic similarity characterizes the similarity of software Causes for semantic similarity include standardization of the software stack in modern enterprises and the popularity of specific types of programming models. As identified in previous work, when enterprises are migrated to the cloud, they are adjusted and standardized so that the same set of agents and processes can be used for management services such as backup recovery, security compliance, and patching [VVK+12]. Images that follow similar operating environment (e.g., Linux distribution or LAMP stack) can be categorized as semantically similar which then enables more efficient and streamlined system administration, troubleshooting, and management.

Figure 4.1(b) depicts the pairwise semantic similarity distribution for the set of images discussed earlier. Here, semantic similarity is the fraction of software in the two images that are semantically similar to each other. Once again we notice the formation of at least two similarity clusters. We also observe that semantic similarity between images from two clusters is higher and there are some images in cluster1 that have high semantic similarity with some images in cluster2 (e.g., images 1 and 16).

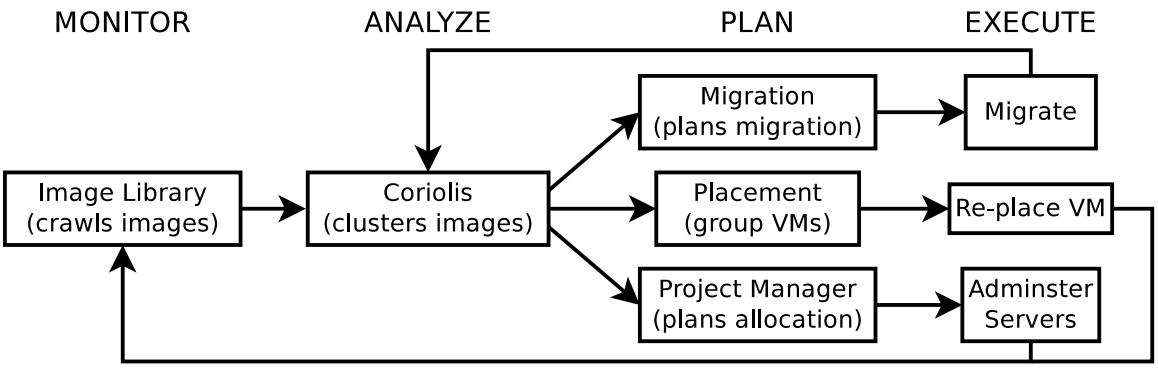


Figure 4.2: **CORIOLIS System Context.**

4.2.3 Harnessing Image Similarity

We identify four common system management scenarios that can leverage image similarity to reduce data center costs. The most important use case for the effects of this thesis is the placement of VMs to hosts or to management systems. Images with high semantic similarity are likely to exhibit higher number of duplicate pages in main memory, which can be deduplicated. Similarly, images with higher content similarity can benefit more from deduplication performed at a shared management server (e.g., vSphere [VMW12]). CORIOLIS analyzes images, which is used by a *planner* to place similar VMs together, improving the effectiveness of underlying deduplication schemes (Figure 4.2).

A second use case is allocation of servers to system administrators for routine maintenance. It has been shown that system administrators can be more efficient and manage up to 80% more servers if the servers have a similar software stack [GP11]. Third, troubleshooting system errors during regular updates in data centers. Troubleshooting in data centers is often akin to manual outlier detection where the engineer attempts to identify servers that responded similarly to the update. Once similar servers are identified, the engineer identifies the difference between the failed server and the successful server to fix the issue. Automated clustering of servers based on semantic similarity can aid such identification.

Use Case	Content	Semantic
VM Placement	✔	✔
Administrator Allocation	✗	✔
Troubleshooting	✗	✔
Migration	✔	✔

Table 4.1: **Similarity types relevant for each use case.**

The final use case is migration of enterprise applications from one data center to another. Migration is performed in batches or waves, where a certain number of images (e.g., 25) are migrated in one weekend [VVK+12]. Migrating images with similar content together can reduce migration time using deduplication. Further, images with similar applications can be reconfigured with fewer application experts, reducing migration cost. Identifying image clusters with both high content and semantic similarity and using them to create waves can help reduce both migration time and cost. Table 4.1 summarizes the type of similarity relevant for all the use cases.

4.3 Similarity-based VM Clustering

Clustering is a well-studied problem in computer science. While the problem is NP-hard, various heuristics exist with acceptable clustering performance. In this section, we examine the popular clustering techniques and study their applicability for VM clustering.

4.3.1 A Representative Clustering Algorithm

k-means is one of the most popular clustering techniques employed in the real world. The algorithm starts with an initial set of k-clusters and refines them iteratively. Even though multiple variants of the algorithm exist, they all apply two canonical operations in each iteration:

- *Assignment Step:* Assign each element to the cluster with the closest mean. $Distance$ computation is the core internal operation, performed k times for each element. If there are N elements to cluster, this requires kN $Distance$ operations.
- *Update step:* Calculate the new mean for each cluster. The core step is a $Merge$ operation which computes the average for 2 elements along each of the D dimensions. In each iteration, across the k clusters, $N - 1$ merge operations are performed.

The worst case time for k-means is exponential in N. For arbitrary set of points in $[0, 1]^D$, if each point is independently perturbed by a normal distribution with variance σ^2, then the expected running time of k-means algorithm is bounded by $O(N^3 4k^3 4D^8 \log^4(N)/\sigma^6)$ [AMR09]. Even for simple cases, the best known bounds on average running time are at least $O(N^4)$.

4.3.2 A Similarity Function for Images

In spite of its high computational complexity in number of elements, k-means is popular in practice because the time taken for each $Distance$ and $Merge$ operation is usually very small. Even for problems with 100 dimensions, $Distance$ and $Merge$ operations require only about 100 addition and division operations. However, these operations are not very well-defined for VM images. We first define a natural definition of these operations and then present the time taken for each operation.

For VM images, it is more natural to define a *similarity* measure than a distance measure. Two images are similar if they contain a large number of identical elements (files or software). Given a pair of images I_i, I_j, similarity between the images can be defined as

$$SIM(I_i, I_j) = \frac{wt(I_i \cap I_j)}{wt(I_i \cup I_j)} \tag{4.1}$$

where $I_i \cup I_j$ is a meta-image that consists of the union of I_i and I_j, $I_i \cap I_j$ is a meta-image that consists of the intersection of I_i and I_j. The weight (wt) function is defined

Image Size	Similarity	Merge
8.8 GB	45.5 sec	14.7 sec
12.3 GB	75.2 sec	24.1 sec
13.6 GB	98.5 sec	31.2 sec
16.3 GB	142.3 sec	44.2 sec
19.7 GB	172.2 sec	53.5 sec
22.1 GB	232.7 sec	64.9 sec

Table 4.2: **Time for Similarity and Merge operations.** Images and file are stored in a database making use of appropriate indices for these operations.

based on the type of similarity that needs to be computed. To estimate content similarity, the wt function is the sum of all files in the image, weighted by the sizes of the files. To estimate semantic similarity, the wt function is the sum of all software deployed in the image weighted by the complexity of the software. Adopting other notions of similarity is straightforward (e.g., a weighted composition of content and semantic similarity). $Distance$ can now be calculated simply as $1 - SIM(I_i, I_j)$. The $Merge$ operation would create a new image that constitutes the set of all unique elements across the images.

4.3.3 Scaling Challenge

We measured the running time for a single $Similarity$ and a single $Merge$ operation on a dual-core 2 GHz Intel Xeon with 4GB memory and images stored on a 5-disk RAID5 SATA array. Table 4.2 lists run times for real images of different sizes. While the actual times seem small, in aggregate, the costs of these operations present a significant challenge. For example, a data center with 1000 images would have to perform 1000^3 similarity computations (even for the best special cases on average complexity), and would need about 2000 years. Computing similarity for images is very expensive because we need to deal with a very large number of dimensions. For content similarity, every unique file in a collection of images is a dimension, with a value of 1 indicating its presence and 0 indicating

its absence in a particular image. The number of dimensions is of the order of millions or more for large images or large collections.

In-memory data structures can reduce the cost of these operations. We conducted experiments by enabling the in-memory feature in MySQL. We observed that the maximum time taken for one similarity computation is 5 seconds (a reduction of $50X$), which though significant only brings down the similarity computation in our previous example to 40 years. Further, this requires the entire index to be memory resident which is not practical. One could envision computing similarity based on only files that are larger than a certain threshold size in each image, but that again would bring down the running time only by a constant factor, while compromising accuracy.

An alternate approach to speed up clustering is to perform approximate clustering based on pair-wise similarity information. The k-medoids clustering algorithm [RK87] does exactly that by restricting the cluster center in an iteration to one of the existing points (images). Hence, both assignment and update steps in each iteration can leverage pair-wise similarity values that are computed in advance. This simplifying approximation, however, still requires pair-wise similarity computation for all images. Since individual similarity operations are expensive for VM images, this approach becomes un-affordable in practice for moderate to large numbers of VMs as is typical in a cloud, as we shall demonstrate later (§6.4). Anecdotally, in a recent study on VM image similarity, the authors reported pair-wise similarity only for a fraction of their image corpus citing scalability challenges [JPZ+11]. With 1000 images, this would take 2 years with the file systems on disk and 15 days with an in-memory system. Clearly, there is a need to reduce the number of operations even further. Unfortunately, k-medoids suffers from an additional challenge, that of determining k a priori. The value of k should ideally be the minimum number of clusters required subject to cluster size constraints dictated by the application. However,

this information is not always known a priori. In the next section, we discuss an approach that successfully overcomes the core limitations of existing clustering approaches.

4.4 CORIOLIS

CORIOLIS uses a novel approach to VM clustering. We discuss this approach and evaluate its scalability relative to the state-of-the-art k-medoids clustering in this section.

4.4.1 Solution Idea: Asymmetric Clustering

To solve the computational and memory challenge in VM clustering, we draw on a key insight in CORIOLIS. First, we observe that to significantly speed-up the $Distance$ and $Merge$ operations, caching only a small subset of the image manifest and hash index of image content must be able to satisfy a large fraction of operations. Enabling cache effectiveness requires introducing asymmetry into the clustering algorithm, that is, the algorithm cannot afford to consider all content from all images as equally important. The CORIOLIS clustering approach involves constructing a tree, where each node in the tree is either a cluster of images or a single image, such that each level in the tree from the root node represents a minimum extent of similarity within images in a cluster. The salient aspects of this approach are:

- **Hierarchical multi-level similarity**: Use multiple levels of similarity to quickly find most relevant clusters. By design, restrict comparisons only with clusters that are similar, reducing the total number of $Similarity$ operations.
- **Ordered Index Lookup**: Clusters at low similarity levels are more popular than leaf nodes. Images with popular content will require more accesses and can be cached.
- **Online Clustering:** Add a new node to existing clusters. Allows addition/deletion of images with only incremental computation.

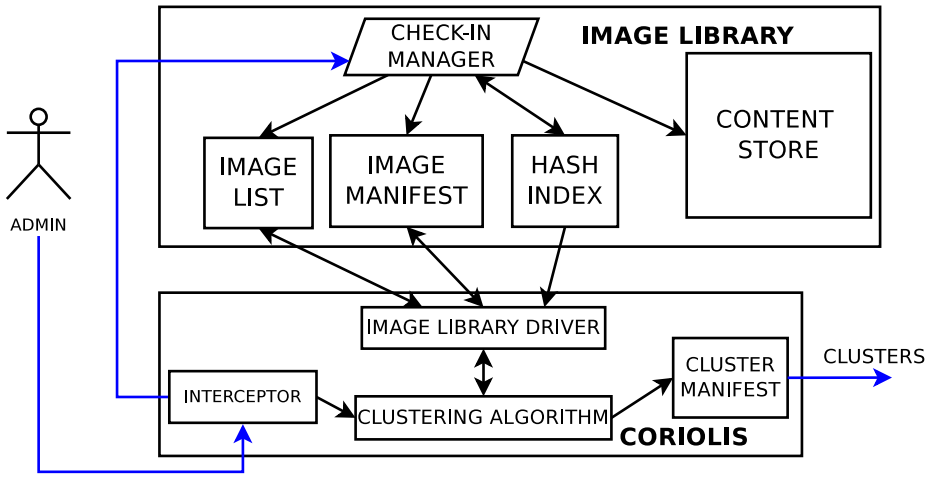


Figure 4.3: **CORIOLIS architecture.** CORIOLIS intercepts image check-in requests and triggers the check-in as well as cluster creation. The clusters are stored in a manifest and can be used by any cloud management components.

4.4.2 CORIOLIS Architecture

In VM image libraries (e.g., [ABM+11]), an image check-in request is intercepted by a *Check-in Manager*, which populates the image list with the new image. Further, an image manifest is created with a list of all the content in the image. For each indexed content, a hash entry is generated and added to a hash index. Duplicate contents are linked to the hash index without generating duplicates. The actual content is stored in a content store, which is linked by the hash index.

CORIOLIS intercepts the check-in request and invokes a *Clustering Algorithm* in addition to the *Check-in Manager*. *Clustering Algorithm* uses an *Image Library Driver* to obtain any relevant information about the images. It then executes a clustering algorithm and generates a *Cluster Manifest* that contains information about the various clusters. The *Cluster Manifest* can be queried by any cloud management component.

The CORIOLIS architecture allows plugging-in any clustering algorithm without changes to the system. The overall image manipulation framework is online in nature but existing clustering algorithms are not online. Further, the clustering technique also needs to solve

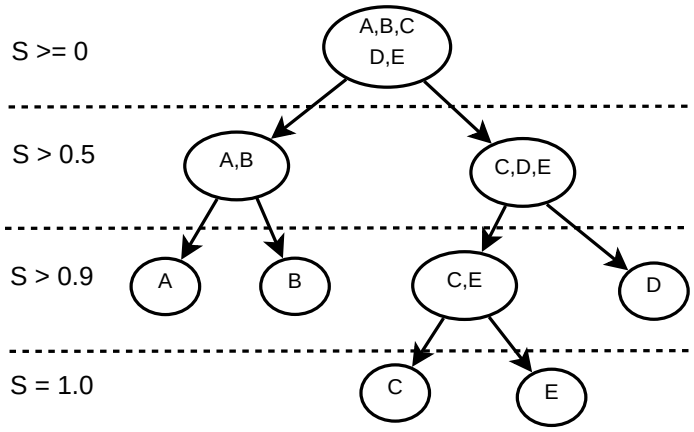


Figure 4.4: **Tree-based clustering.** Computed Similarity Values {(A,B):0.75, (C,E):0.95, (CE, D):0.8}.

the additional problems of scale and expensive cluster center computation. Finally, an ideal image clustering system needs to ensure that the time taken to compute the clusters should be a small fraction of the time taken to check-in an image. The clustering technique that we describe next is designed to address these practical challenges.

4.4.3 CORIOLIS' Tree-based Clustering

CORIOLIS's tree-based clustering approach is outlined below and it is based on two key ideas. The most common operation in clustering is to identify the cluster most similar to a given element and the first idea focuses on speeding up this operation. Since clusters can grow to become very large whereas individual images are typically small, we define and use an asymmetric similarity function S within CORIOLIS that runs in time proportional to the smaller of the two. In particular, we define similarity as the coverage offered by a larger node B (typically a cluster) to a new node A that is being added to the cluster by replacing the $union$ operator in the denominator by the min operator.

$$S = \frac{wt(A \cap B)}{min(wt(A), wt(B))} \tag{4.2}$$

Our second key idea is to ensure skew in the usage of images and image clusters allowing effective caching. Further, we reuse the similarity computations done for an image when computing similarity for other images. CORIOLIS uses a tree-based partitioning of the images to achieve both these goals. Each level of the tree represents a predefined minimum level (extent) of similarity. The root of the tree captures a similarity level $S \geq 0$. Thus, all images can be clustered in this meta-node. The last level of the tree captures a similarity level $S = 1$; it consists of either single images or a collection of duplicate images. Intermediate levels represent predefined similarity levels, $0 < S < 1$, which increases with the depth of the tree. We elaborate our representation using the example in Figure 4.4. Consider 5 images A, B, C, D, E. The tree has 4 levels representing similarity of $0, 0.5, 0.9$ and 1 respectively. A and B have a similarity measure of 0.75. Hence, they are clustered at level $S > 0.5$ but are independent nodes at level $S > 0.9$. Similarly, C and E have a similarity of 0.95 and are grouped together up to all levels $S > 0.9$ but are independent nodes at level $S = 1$.

Given a new image v_i, our goal is to find similar nodes (or meta-nodes) with as few $Similarity$ operations as possible. CORIOLIS's grouping of VM image clusters within a hierarchical tree structure allows early pruning of images that are not similar to the new image v_i. Adding a new image to the CORIOLIS VM image tree, the new image is first added to the root meta-node. Once an image is added to a node, we compute the similarity of the new node with each of its children to determine if it can be added to any child. If the similarity S level is found adequate with more than one child, the new image is only added to the child node with which the similarity is the greatest. If no such child node exists, we create a new child node and add v_i to the node. This process terminates when we reach a leaf node.

Figure 4.5 illustrates a new image F as it traverses the tree. It is important to note here that the number of $Similarity$ and $Merge$ operations executed for an image is proportional

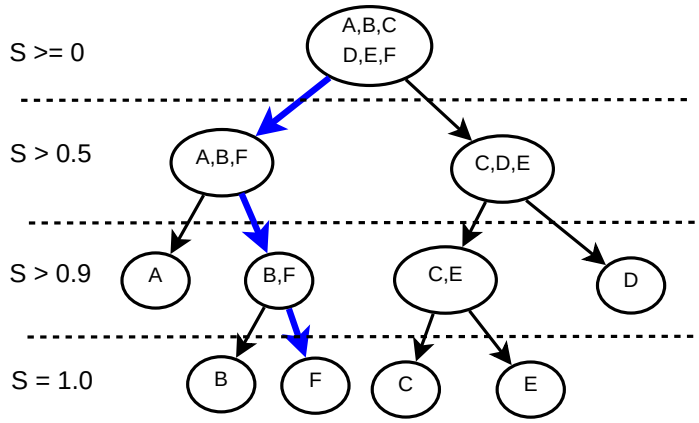


Figure 4.5: **Clustering a new image F.** Computed Similarity Values are {(AB,F):0.95, (CDE,F):0.3, (A,F):0.75, (B:F):0.95}.

to the depth of the tree. The depth of the tree is a pre-defined constant, bound by the log of the number of images inserted. Hence, the approach allows us to create a tree in time no more than $O(N \log N)$, where N is the number of images. And given the similarity levels at various tree depths, the tree can then be queried in linear time for clusters with specific properties.

4.4.4 Scalability Evaluation

To evaluate CORIOLIS, we used VM images from 2 production data centers. The first set of 9 images is from a large-scale enterprise data center at IBM. The latter set of 12 images is from the CS department's small-scale data center at Florida International University. The former set of images are diverse compared to the latter set reflecting the needs typical of a large-scale enterprise data center. Next, we created increasingly larger sets of images from these initial set of 21 production images. We did this by separating out 3 of the 21 images and randomly sampling files contained within these to generate synthetic images. The net effect is that the synthetic images contain a random combination of files from these 3 source images. We performed clustering experiments in a Linux VM configured with

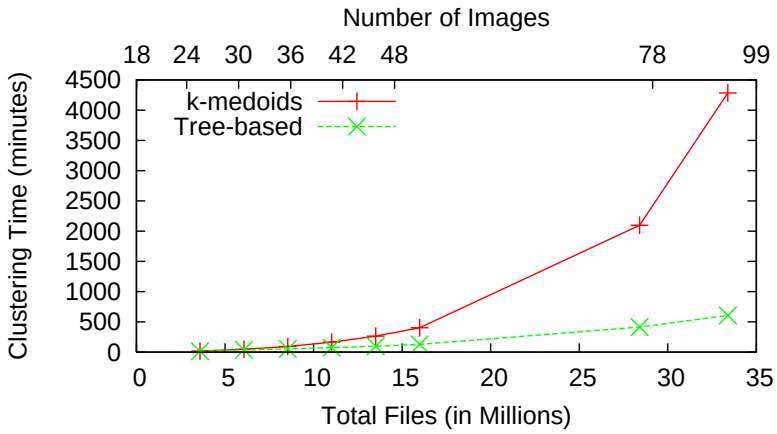


Figure 4.6: **Scalability of k-medoids and CORIOLIS' tree-based clustering algorithms.**

16 GB RAM on an 6-core AMD Opteron processor virtualized using the VMware ESX hypervisor.

We choose k-medoids for this comparison as it is significantly faster than k-means. Fig 4.6 presents the time taken by the k-medoids algorithm and the tree-based clustering algorithm as the problem size is increased. The time includes the time taken to read file metadata and store it in a database, where similarity and merge operations are performed. The k-medoids algorithm takes significantly longer and displays a quadratic increase in clustering time as the number of images is increased. We observed that more than 95% of the time is spent in computing similarity as the cluster size is increased. For clustering 99 images, it takes nearly 3 days, which is clearly unacceptable. In contrast, our tree-based clustering algorithm reduces the number of similarity computations by a factor of 8 and is able to cluster the images within 10 hours; an acceptable window of time even for heavyweight management VM tasks carried out over weekends.

4.5 Summary

We described the CORIOLIS framework and system that was specifically designed for scalable clustering of VM images to intelligently place VM into hosts, and with the aid of dedu-

plication techniques, optimize memory usage through sharing and overall system performance. We argued that the state-of-the-art k-medoids clustering algorithm incurs quadratic complexity which we demonstrated as infeasible for cloud scale data centers. CORIOLIS's distinguishing strength lies in its scalable tree-based image clustering technique that supports an arbitrary similarity metric. This novel technique allows clustering to be performed in $O(N \log N)$ time for a data center with N images, allowing it to scale to large data centers.

In the next chapters we propose systems which improve on how cache accesses, in particular file writes, are handled when using the operating system cache.

4.6 Credits

This work was first published in the proceedings of the International Conference on Autonomic Computing in June 2015 [CCV+13] and was presented by Daniel Campello. All authors contributed to the design of CORIOLIS. Daniel Campello and Carlos Crespo implemented CORIOLIS in Linux. Daniel Campello refined the implementation to its final state. Daniel Campello executed all the experiments to evaluate the system.

CHAPTER 5

NON-BLOCKING WRITES TO FILES

In Chapter 4 we proposed a solution that aims to improve cache hit rates and memory usage in virtualized systems to improve overall system performance. In this chapter we center our efforts inside the operating system (OS) and we propose solutions that make a more efficient use of caching by altering the existing way of handling write accesses.

While blocking the process for a page fetch cannot be avoided in case of a read to a non-cached page, it can be entirely eliminated in case of writes. The OS could buffer the data written temporarily elsewhere in memory and unblock the process immediately; fetching and updating the page can be performed asynchronously. This decoupling of page write request by the application process from the OS-level page update allows two crucial performance enhancements. First, the process is free to make progress without having to wait for a slow page fetch I/O operation to complete. Second, the parallelism of page fetch operations increases; this improves page fetch throughput since storage devices offer greater performance at higher levels of I/O parallelism.

In this chapter, we explore new design alternatives and optimizations for non-blocking writes, address consistency and correctness implications, and present an implementation and evaluation of these ideas. By separating page fetch policy from fetch mechanism, we implement and evaluate two page fetch policies: *asynchronous* and *lazy*, and two page fetch mechanisms: *foreground* and *background*. We also develop *non-blocking reads* to recently written data in non-cached pages.

We implemented non-blocking writes to files in the Linux kernel. Our implementation works seamlessly inside the OS requiring no changes to applications. We integrate the handling of writes to non-cached file data for both local file systems and network file system clients within a common design and implementation framework. And because it builds on

a generic design, our implementation provides a starting point for similar implementations in other operating systems.

5.1 Motivating Non-blocking Writes to Files

Previous studies that have analyzed production file system workloads report a significant fraction of write accesses being small or unaligned writes [ELMS03, LPGM08, ODCH+85, RA00]. Technology trends also indicate an increase in page fetch rates in the future. On the server end, multi-core systems and virtualization now enable more co-located workloads leading to larger memory working sets. As the effective memory working sets [Den68, KVR10] of workloads continue to grow, page fetch rates also continue to increase. A host of flash-based hybrid memory systems and storage caching and tiering systems have been inspired, and find relevance in practice, because of these trends [BLM+12, CKZ11, EMC12, Fus12, GMC+12, GPG+11, KM06, KMR+13, Net13, PVO+15, SS10a, VMw13, WR10]. On the personal computing end, newer data intensive desktop/laptop applications place greater I/O demands [HDV+11]. In mobile systems, page fetches have been found to affect the performance of the data-intensive applications significantly [KAU12].

5.1.1 Addressing the fetch-before-write problem

Non-blocking writes eliminate the fetch-before-write requirement by creating an in-memory patch for the updated page and unblocking the process immediately. This modification is illustrated in Figure 5.1.

Reducing Process blocking

Processes block when they partially overwrite one or more non-cached file pages. Such overwrites may be of any size as long as they are not perfectly aligned to page boundaries. Non-blocking writes reduces process blocking by eliminating the synchronous page

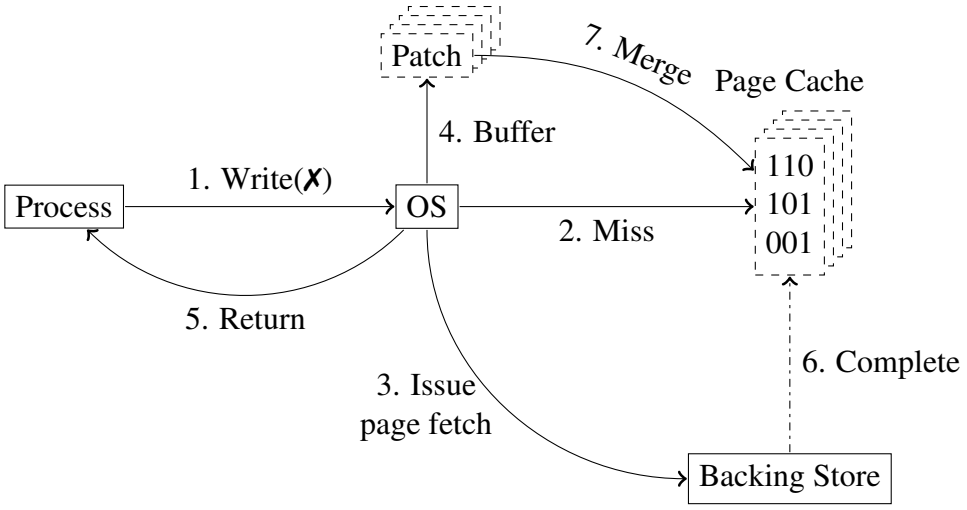


Figure 5.1: **A non-blocking write employing asynchronous fetch.** The process resumes execution (Step 5) after the patch is created in memory while the originally blocking I/O completion is delayed until later (Step 6). The dash-dotted line represents a slow transition.

fetch latency for all writes and many reads to pages missing in the page cache. Previous studies have reported about the significant fraction of small or unaligned writes in production file system workloads [ELMS03, LPGM08, ODCH+85, RA00]. However, little is known about partial page overwrite behavior. To better understand the prevalence of such file writes in production workloads, we developed a Linux kernel module that intercepts file system operations and reports sizes and block alignment for writes. We then analyzed one day's worth of file system operations collected from several production machines at Florida International University's Computer Science department. Besides these we also analyzed file system traces of much shorter duration (two minutes each) available in MobiBench [ESO13, JLH+13]. Table 5.1 provides a description of all the traces we analyzed.

Figure 5.2 provides an analysis of the write traffic on each of these machines. On an average, 63.12% of the writes involved partial page overwrites. Depending on the size of the page cache, these overwrites could result in varying degrees of page fetches prior to the page update. The degree of page fetches also depends on the locality of data accesses in the workload wherein a write may follow a read in short temporal order. To account for access

Workload	Description
ug-filesrv	Undergrad NFS/CIFS fileserver
gsf-filesrv	Grad/Staff/Faculty NFS/CIFS fileserver
moodle	Web & DB server for department CMS
backup	Nightly backups of department servers
usr1	Researcher 1 desktop
usr2	Researcher 2 desktop
Facebook	MobiBench Facebook trace [ESO13]
twitter	MobiBench twitter trace [ESO13]

Table 5.1: **Workloads traced and their descriptions.**

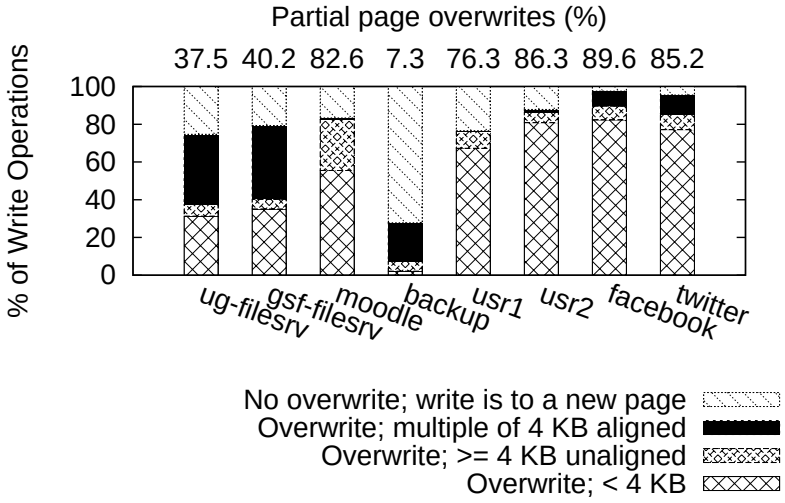


Figure 5.2: **Breakdown of write operations by amount of page data overwritten.** Each bar represents a different trace and the number above each bar is the percentage of write operations than involve at least one partial page overwrite.

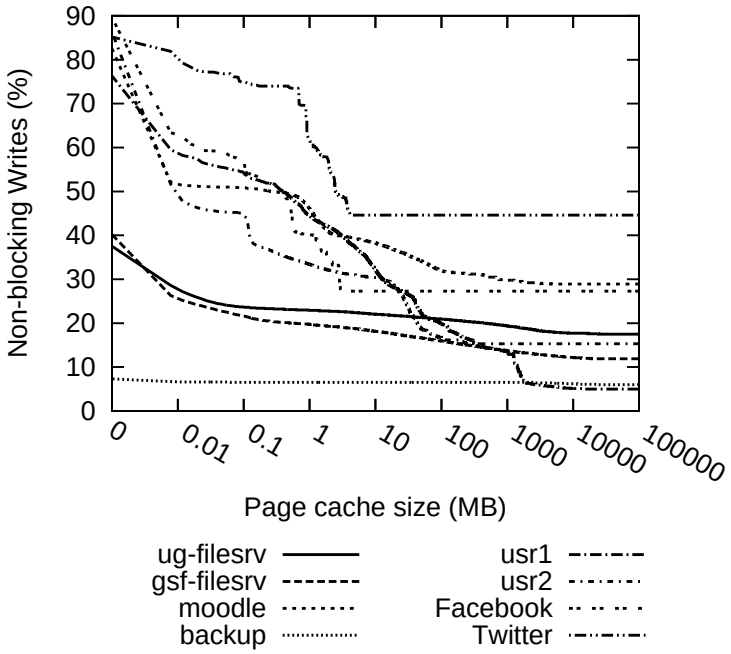


Figure 5.3: **Non-blocking writes as a percentage of total write operations.** Varying the page cache size in the x-axis.

locality, we refined our estimates using a cache simulator to count the number of writes that actually lead to page fetches at various memory sizes. Such writes can be made non-blocking. The cache simulator used a modified Mattson's LRU stack algorithm [MGST70] and uses the observation that a non-blocking write at a given LRU cache size would also be a non-blocking write at all smaller cache sizes. Modifications to the original algorithm involved counting all partial page overwrites to pages not in the cache as non-blocking writes. Figure 5.3 presents the percentage of total writes that would benefit from non-blocking writes for the workloads in Table 5.1. For a majority of the workloads, this value is at least 15% even for a large page cache of size 100GB. A system that can make such writes non-blocking would make the overall write performance less dependent on the page cache capacity.

5.1.2 Addressing Correctness

With non-blocking writes, the ordering of read and write operations within and across processes in the system are liable to change. As we shall elaborate later (§5.2.2), the *patch creation* and *patch application* mechanisms in non-blocking writes ensure that the ordering of causally dependent operations is preserved. The key insights that we use are: *(i)* reads to recent updates can be served correctly using the most recently created patches, *(ii)* reads that block on page-fetch are allowed to proceed only after applying all the outstanding patches, and *(iii)* reads and writes that are independent and issued by the same or different threads can be reordered without loss of correctness.

Another potential concern with non-blocking writes is data durability. For file data, we observe that the asynchronous write operation only modifies volatile memory and the OS makes no guarantees that the modifications are durable. With non-blocking writes, synchronous writes (on account of `sync/fsync` or the periodic page-flusher daemon) block to wait for the required fetch, apply any outstanding patches, and write the page to storage before unblocking the process. Thus, the durability properties of the system remain unchanged with non-blocking writes.

5.2 Approach Overview

The page fetch process blocks process execution, which is undesirable. Non-blocking writes work by buffering updates to non-cached pages by creating *patches* in OS memory to be applied later. A non-blocking write returns immediately once a patch of the update is created and queued to the list of pending page updates. The page only becomes *Up-to-date* once all the pending patches are applied after its fetch is completed.

Non-blocking writes alter write control flow, thus affecting reads to recently written data. Further, they require managing additional cached data in the form of patches. The rest of this section discusses these details.

5.2.1 Write Handling

Operating systems allow writes to file data via two common mechanisms: supervised *system calls* and unsupervised *memory mapped access*.

To handle supervised writes, the OS uses the system call arguments — the address of the data buffer to be written, the size of the data, and the file (and implicitly, the offset) to write to — and resolves this access to a data page write internally. With non-blocking writes, the OS extracts the data update from the system call arguments, creates a patch, and queues it for later use.

Unsupervised file access can be provided by memory mapping a portion of a file to the process address space. In our current design, memory mapped access are handled as in current systems by blocking the process to service the page fault.

5.2.2 Patch Management

Since commodity operating systems handle data at the granularity of pages, we chose a design where each patch will apply to a single page. We abstract an update with a *patch* data structure that contains the data to be written along with its target location and size. To handle multiple overwrites to the same page, we implement *per-page patch queues* wherein page patches are queued and later applied to the page in FIFO order. When new data is adjacent or overwrites existing patches, it is merged into existing patches accordingly. This makes patch memory overhead and patch application overhead proportional to the number

of page bytes changed in the page instead of the number of bytes written to the page since the page was last evicted from memory.

When a page is read in either via a system call induced page fetch or a memory-mapped access causing a page fault, the first step is to apply outstanding patches, if any, to the page to bring it up-to-date before the page is made accessible. Patches are applied by simply copying patch data to the target page location. Patch application occurs in the bottom-half interrupt handling of the page read completion event (further discussed in §5.4). If any patches are applied, the page flag indicating that the page is dirty is set so that the page is correctly written to the backing store when durability is requested.

5.2.3 Non-blocking Reads

Similar to writes, reads can be classified as *supervised* and *unsupervised* as well. Reads to non-cached pages block the process in current systems. With non-blocking writes, a new opportunity to perform *non-blocking reads* becomes available. Specifically, if the read is serviceable from the patches queued on the page, then the reading process can be unblocked immediately without incurring a page fetch I/O by copying the data from patches into the target buffer. The read is not serviceable if any portion of the data being requested is not contained within the patch queue. In such a case, the reading process blocks for the page to be fetched. For unsupervised reads, our current design blocks the process for the page fetch in all cases.

5.3 Alternative Page Fetch Modes

We now explore the page fetch modes that become possible with non-blocking writes. These variants highlight the opportunities for further optimizing resource consumption and improving performance enabled by non-blocking writes.

5.3.1 Asynchronous Page Fetch (NBW-Async)

In this mode, page fetch I/O is queued to be issued at the time of the page write. The appeal of this approach is its simplicity.

Asynchronous page fetch defines policy. However, its mechanism may involve additional blocking prior to issuing the page fetch. We discuss two alternative page fetch mechanisms that highlight this issue.

1. **Foreground Asynchronous Page Fetch (NBW-Async-FG)**. The page fetch I/O is issued in the context of the process performing the write to the file page. Our discussion in previous sections was based on this mechanism. Although the process does not wait for the completion of the data fetch, issuing the fetch I/O for the data page may itself involve retrieving additional metadata pages to locate the data page if these metadata pages are not cached in OS memory. If so, the writing process would have to block for the necessary metadata fetches to complete, thereby voiding most of the benefits of the non-blocking write.
2. **Background Asynchronous Page Fetch (NBW-Async-BG).** The writing process moves all work necessary to issue the page fetch to a different context by using kernel worker threads. This approach eliminates any blocking of the writing process owing to metadata misses; a worker thread blocks for all fetches while the issuing process continues its execution.

Asynchronous fetch is a valuable improvement. However, it consumes system resources, allocating system memory for the page to be fetched and using storage I/O bandwidth to fetch the page.

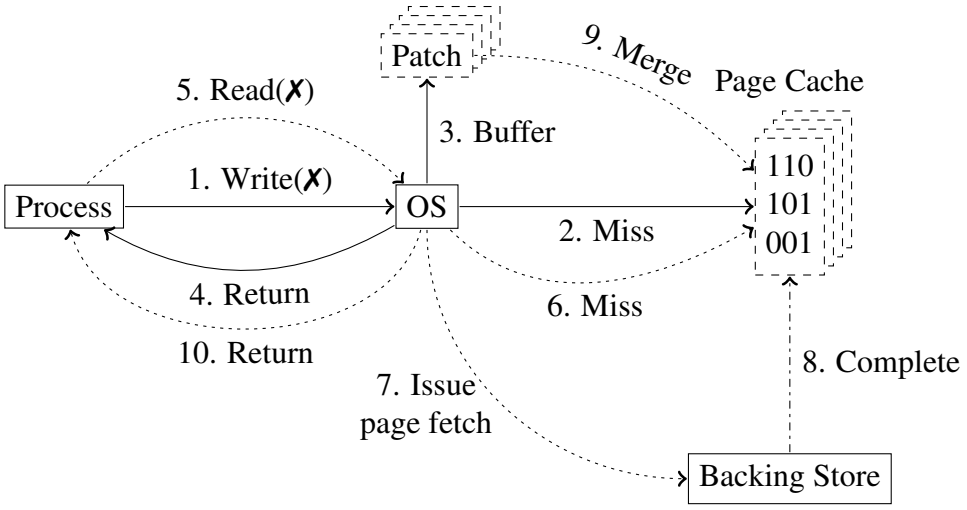


Figure 5.4: **A non-blocking write employing lazy fetch.** The process resumes execution (Step 4) after the patch is created in memory. The Read operation in Step 5 optionally occurs later in the execution while the originally blocking I/O is optionally issued and completes much later (Step 8). The dash-dotted arrow represents a slow transition.

5.3.2 Lazy Page Fetch (NBW-Lazy)

When a process writes to a non-cached data page, its execution is not contingent on the page being available in memory. With *lazy page fetch*, the OS delays the page fetch until it becomes unavoidable. Lazy page fetch has the potential to further reduce the system's resource consumption. Figure 5.4 illustrates this alternative.

Lazy page fetch creates new system scenarios which must be considered carefully. If a future page read cannot be served using the currently available patches for the non-cached page, the page fetch becomes unavoidable. In this case, the page is fetched synchronously and patches are applied first before extracting the data to service the read operation. If the page gets overwritten in its entirety or if page persistence becomes unnecessary for another reason (e.g., the containing file is deleted), the original page fetch is eliminated entirely.

Page data durability can become necessary in the following instances: *(i)* synchronous file write by an application, *(ii)* periodic flushing of dirty pages by the OS [Bac86], or *(iii)* ordered page writes to storage as in a journaling file system [Hag87, PADAD05]. In all

these cases, the page is fetched synchronously before being flushed to the backing store. Lastly, non-blocking writes are not engaged for metadata pages which use the conventional durability mechanisms. Durability related questions are discussed further in §5.4.2.

5.4 Implementation

We present an overview of the implementation of non-blocking writes and discuss details related to how it preserves system correctness.

5.4.1 Overview

We implemented non-blocking writes for file data in the Linux kernel (version 2.6.34.17) by modifying the generic virtual file system (VFS) layer. Unlike the conventional Linux approach, all handling of fetch completion (such as applying patches, marking the page dirty, processing a journaling transaction, and unlocking the page) occurs in the bottom-half I/O completion handler.

5.4.2 Implementation Insights

Journaling File Systems. Our implementation of non-blocking writes preserves the correctness of journaling file systems by allowing the expected behavior for various journaling modes. For instance, non-blocking writes preserve ext4's ordered mode journaling invariant that data updates are flushed to disk before transactions containing related metadata updates. Metadata transactions in ext4 do not get processed until after the related data page is fetched into memory, outstanding patches are applied, the page is marked dirty, and dirty buffers added to the transaction handler. Thus, all dirty data pages related to a metadata

transaction are resident in memory and flushed to disk by ext4's ordered mode journaling mechanism prior to committing the transaction.

Ordering of Operations to the *Same* Page. While a non-blocking write is being handled within the operating system, multiple operations such as read, prefetch, synchronous write, and flush, can be issued to the page involved. Operating systems carefully synchronize these operations to maintain consistency and return only up-to-date data to applications. Our implementation respects the Linux page locking protocol. A page is locked after it is allocated and before issuing a fetch for it. As a result, kernel mechanisms such as fsync and mmap are also supported correctly. These mechanisms block on the page lock which becomes available only after the page is fetched and patches applied before proceeding to operate on the page. When delayed page fetch mechanisms (as in NBW-Async-BG and NBW-Lazy) are used, an `NBW` entry for the page involved is added in the page cache mapping for the file before the page is allocated. This `NBW` entry allows for locking the page to maintain the ordering of page operations. When necessary (e.g., a *sync*), pages indexed as `NBW` get fetched which in turn involves acquiring the page lock, thus synchronizing future operations on the page. The only exception to such page locking is writing to a page already in the non-blocking write state; the write does not lock the page but instead queues a new patch.

Ordering of Operations to *Different* Pages. Non-blocking writes may alter the sequence in which patches *to different pages* get applied since the page fetches may complete out-of-order. Non-blocking writes only replace writes that are to memory which are not guaranteed to be reflected to persistent storage in any particular sequence. Thus, ordering violations in updates of in-memory pages are crash-safe.

Flushing to Storage. If an application would like explicit disk ordering for memory page updates, it would execute a blocking flush operation (e.g., `fsync`) subsequent to each operation. The flush operation causes the OS to force the fetch of any page indexed as `NBW`

even if it has not been allocated yet. The OS then obtains the page lock, waits for the page fetch, and applies any outstanding patches, before flushing the page and returning control to the application. Ordering of disk writes are thus preserved with non-blocking writes.

Multi-core and Kernel Preemption. Our implementation fully supports SMP and kernel preemption. For a given non-cached page, the patch creation mechanism (when processing the write system call) can contend with the patch application mechanism (when handling page fetch completion). Our implementation uses a single additional lock to protect a patch queue from simultaneous access.

5.5 Evaluation

We address the following questions:

(1) What are the benefits of non-blocking writes for different workloads?

(2) How do the fetch modes of non-blocking writes perform relative to each other?

(3) How sensitive are non-blocking writes to the underlying storage type?

(4) How does memory size affect non-blocking writes?

We evaluate four different solutions. Blocking writes (*BW*) is the conventional approach to handling writes and uses the Linux kernel implementation. Non-blocking writes variants include asynchronous mode using foreground (*NBW-Async-FG*) and background (*NBW-Async-BG*) fetch, and lazy mode (*NBW-Lazy*).

Workloads and Experimental Setup.

We use the Filebench micro-benchmark [SUN11] to address (1), (2), (3), and (4) using controlled workloads. We use the SPECsfs2008 benchmark [Sta08] and replay the MobiBench traces [ESO13] to further analyze questions (1) and (2); the MobiBench trace replay also helps answer question (3). The Filebench and MobiBench evaluations were

performed on a machine with Quad-Core 2.50 GHz AMD Opteron(tm) 1381 processors, 8GB of RAM, a 500 GB WDC WD5002ABYS hard disk-drive, a 32 GB Intel X25-E SSD, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14) . The above setup was also used to run the client-side component of the SPECsfs2008 benchmark. Additionally, for the SPECsfs2008 benchmark, the NFS server used a 2.3 GHz Quad-Core AMD Opteron(tm) Processor 1356, 7GB of RAM, 500 GB WDC and 160 GB Seagate disks, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14). The 500GB hard disk housed the root file system while the 160GB hard disk stored the NFS exported data. The network link between client and server was Gigabit Ethernet.

5.5.1 Filebench Micro-benchmark

For all the following experiments we ran five Filebench personalities for 60 seconds using a 5GB pre-allocated file after clearing the contents of the OS page cache. Each personality represents a different type of workload. The system was configured to use 4GB of main memory and memory used for patches was limited to 64MB, a small fraction of DRAM, to avoid significantly affecting the DRAM available to the workload and the OS. We report the Filebench performance metric, the number of operations per second. Each data-point is calculated using the average of 3 executions.

Performance Evaluation

We first examine the performance of Filebench when using a hard disk as the storage back-end. Figure 5.5 depicts the performance for four Filebench personalities when varying the size of the Filebench operation. Each data point reports the average of 3 executions. Standard error of measurement was less than 3% of the average for 96.88% of the cases and were less than 10% for the rest.

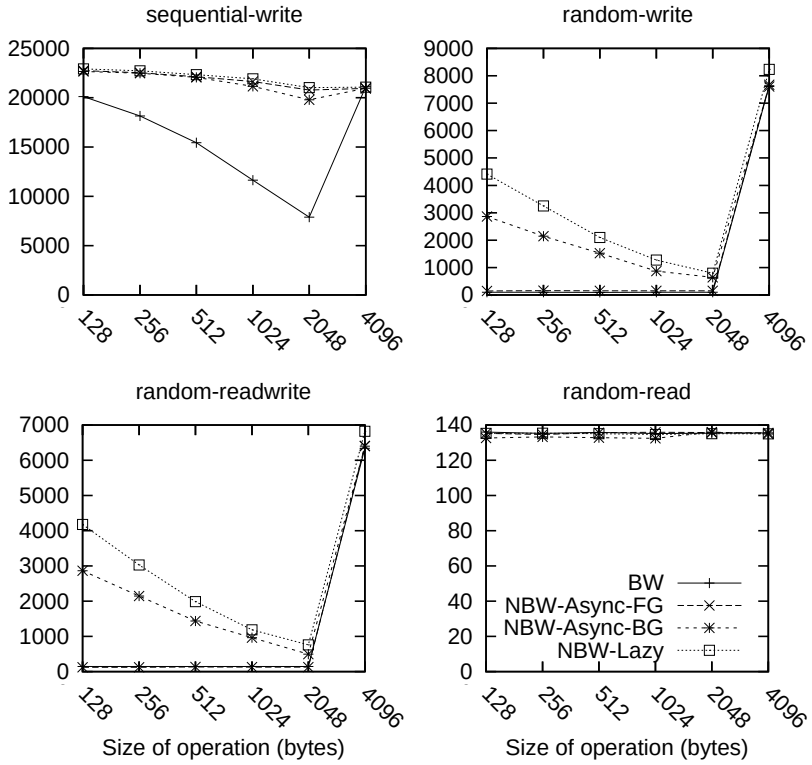


Figure 5.5: **Performance for Filebench personalities when using hard disk-drive.** Varying I/O size on the x-axis.

The first three plots involve personalities that perform write operations. At 4KB I/O size, there is no fetch-before-write behavior because every write results in an overwrite of an entire page; thus, non-blocking writes are not engaged and do not impose any overhead either.

For the *sequential-write* personality, performance with blocking writes (BW) depends on the operation size, and is limited by the number of page misses per operation. In the worst case, when the I/O size is equal to 2KB, every two writes involve a blocking fetch. On average, the different non-blocking write modes provide a performance improvement of 13-160% depending on the I/O size.

The second and third personalities represent random access workloads. *Random-write* is a write-only workload, while *random-readwrite* is a mixed workload; the latter uses two threads, one for issuing reads and the other for writes. For I/O sizes smaller than 4KB, BW

provides a constant throughput of around 97 and 146 operations/sec for *random-write* and *random-readwrite* personalities respectively. Performance is consistent regardless of the I/O size because each operation is equally likely to result in a page miss and fetch. *Random-readwrite* performs better than *random-write* due to the additional available I/O parallelism when two threads are used. Further, for *random-write*, NBW-Async-FG provides 50-60% performance improvement due to reduced blocking for page fetches of the process. However, this improvement does not manifest for *random-readwrite* wherein read operations incur higher latencies due to additional blocking for pages with fetches in progress. In both cases, the benefits of NBW-Async-FG are significantly lower when compared to other non-blocking write modes since NBW-Async-FG blocks on many of the initial file-system metadata misses during this short-running experiment.

In contrast, NBW-Async-BG unblocks the process immediately while a different kernel thread blocks for the metadata fetches as necessary. This mode shows a 6.7x-29.5x performance improvement for *random-write*, depending on the I/O size. These performance gains reduce as the I/O size increases since non-blocking writes can create fewer outstanding patches to comply with the imposed patch memory limit of 64MB. A similar trend is observed for *random-readwrite* with performance improvements varying from 3.4x-19.5x depending on the I/O size used. NBW-Lazy provides up to 45.4X performance improvement over BW by also eliminating both data and metadata page fetches when possible. When the available patch memory limit is reached, writes are treated as in BW until more patch memory is freed up.

The final two personalities, *random-read* and *sequential-read* (not shown), are read-only workloads. These workloads do not create write operations and the overhead of using a non-blocking writes kernel is zero. Non-blocking writes deliver the same performance as blocking writes.

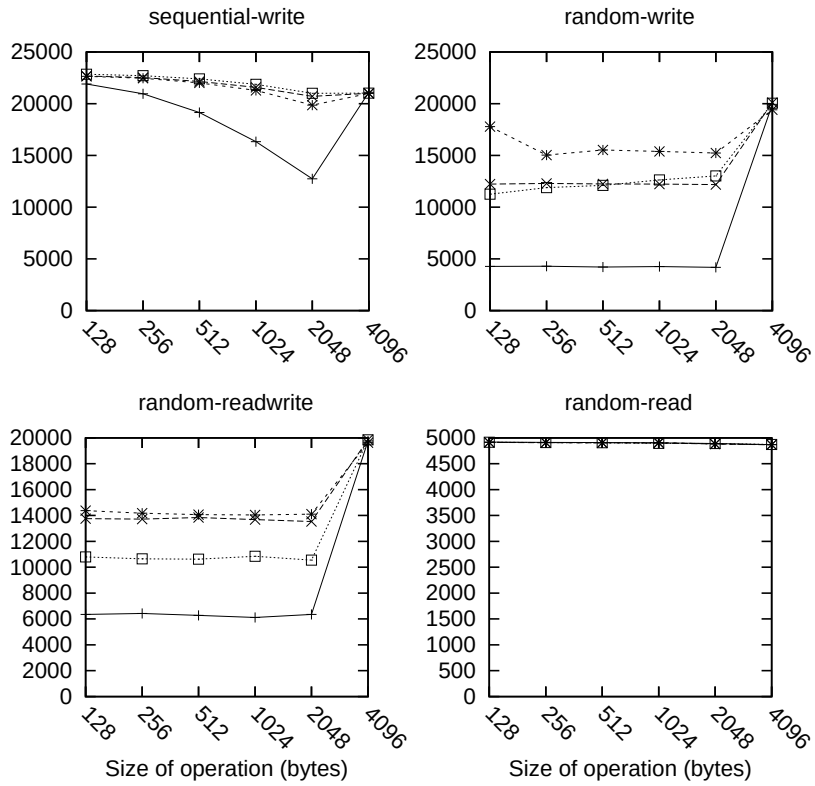


Figure 5.6: **Performance for Filebench personalities when using solid-state drive.** Varying I/O size on the x-axis.

Sensitivity to system parameters

Our sensitivity analysis of non-blocking writes addresses the following specific questions:

(1) What are the benefits of non-blocking writes when using different storage back-ends?

(2) How do non-blocking writes perform when system memory size is varied?

Sensitivity to storage back-ends

To answer the first question, we evaluated non-blocking writes using a solid state drive (SSD) based storage back-end. Figure 5.6 presents results when running Filebench personalities using a solid state drive. Each data point reports the average of 3 executions. Standard error of measurement was less than 2.25% of the average in all cases except one for which it was 5%.

Performance trends with the *sequential-write* workload are almost identical to the hard disk counterparts (Figure 5.5) for all modes of non-blocking writes. This is because non-blocking writes completely eliminate the latency of accessing storage for every operation in both systems. On the other hand, because the SSD offers better throughput than the hard disk drive, BW offers an increase in throughput for every size below 4KB. In summary, the different non-blocking write modes provide between 4% and 61% performance improvement depending on the I/O size.

For the *random-write* and *random-readwrite* workloads, the non-blocking write variants all improve performance but to varying degrees. The SSD had significantly lower latencies servicing random accesses relative to the hard disk drive which allowed for meta-data misses to be serviced much quicker. The efficiency of NBW-Async-FG relative to BW is further improved relative to the hard disk system and it delivers 188% and 117% performance improvement for *random-write* and *random-readwrite* respectively. NBW-Async-BG improves over NBW-Async-FG for reasons similar to those with hard disks. NBW-Async-BG delivers 272% (up to 4.2X in the best case) and 125% performance improvement over BW on average for *random-write* and *random-readwrite* respectively. Lastly, although NBW-Lazy performs significantly better than BW, contrary to our expectations, its performance improvements were lower when compared to the NBW-Async modes. Upon further investigation, we found that when the patch memory limit is reached, NBW-Lazy takes longer than the other modes to free its memory given that the fetches are issued only when blocking cannot be avoided anymore. While the duration of the experiment is the same as disk drives, a faster SSD results in the patch memory limit being met more quickly. In our current implementation, after the patch memory limit is reached and no more patches can be created, NBW-Lazy defaults to a BW behavior issuing fetches synchronously for handling writes to non-cached pages. Despite this drawback, NBW-Lazy mode shows 163%-211% and 70% improvement over BW for *random-write* and *random-readwrite* respectively.

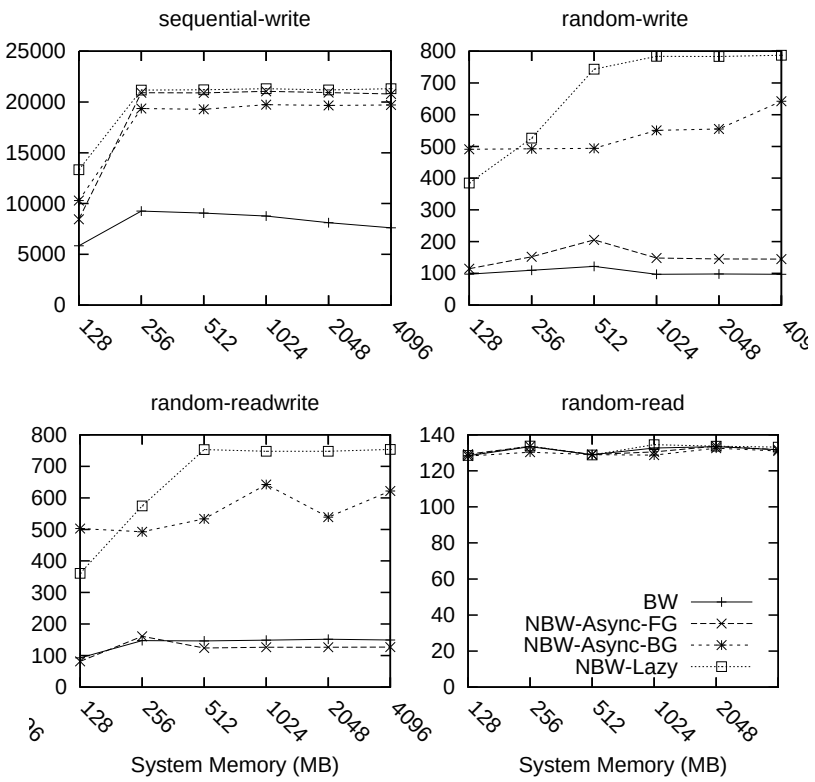


Figure 5.7: **Memory sensitivity of Filebench.** The I/O size was fixed at 2KB and patch memory limit was set to 64MB.

Sensitivity to system memory size

We answer the second question using the Filebench workloads and varying the amount of system memory available to the operating system. For these experiments, we used a hard disk drive as the storage back-end and fixed the I/O size at 2KB. Figure 5.7 presents the results of this experiment. Each data point reports the average of 3 executions. Standard error of measurement was less than 4% of the average for 90% of the cases and were less than 10% for the rest.

For the *sequential-write* workload, the non-blocking writes variants perform 45-180% better than BW. Further NBW-Lazy performs better and can be considered optimal because *(i)* it uses very little patch memory, sufficient to hold enough patches until a single whole

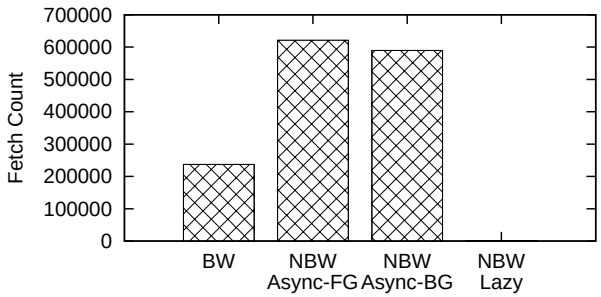


Figure 5.8: **Page fetches issued for the Filebench sequential-write workload.**

page is overwritten, and *(ii)* since pages get overwritten entirely in the sequential write, it eliminates all page fetches.

Figure 5.8 depicts the number of page fetches for the *sequential-write* workload. For BW, NBW-Async-FG, and NBW-Async-BG the number of fetches is proportional to the number of operations per second reported for these runs. On the other hand, NBW-Lazy performs zero fetches.

For *random-write* and *random-readwrite* workloads, NBW-Async-FG delivers performance that is relatively consistent with BW; the I/O performance achieved by these solutions is not high enough to make differences in memory relevant. NBW-Async-BG and NBW-Lazy offer significant performance gains relative to BW of as much as 560% and 710% respectively. With NBW-Lazy, performance improves with more available memory but only up to the point at which the imposed patch memory limit is reached prior to the completion of the execution; increasing the patch memory limit would allow NBW-Lazy to continue scaling its performance.

5.5.2 SPECsfs2008 Macro-benchmark

The SPECsfs2008 benchmark tests the performance of NFS servers. For this experiment, we installed a non-blocking writes kernel in the NFS server which exported the network file system in *async* mode. SPECsfs2008 uses a client side workload generator that bypasses

Operation Size	Read %	Cum. Read %	Write %	Cum. Write %
1 - 511 bytes	3	3	13	13
512 - 1023 bytes	1	4	3	16
1024 - 2047 bytes	2	6	7	23
2048 - 4095 bytes	1	7	5	28
4KB	16	23	11	39
4097 - 8191 bytes	6	29	3	42
8KB	36	65	30	72
8193 - 16383 bytes	7	72	7	79
16KB	7	79	5	84
16385 - 32767 bytes	2	81	1	85
32, 64, 96, 128, 256 KB	19	100	15	100

Table 5.2: **SPECsfs2008 write sizes.**

the page cache entirely. The client was configured for a target load of 500 operations per second. The target load was sustained in all evaluations; thus the SPECsfs2008 performance metric is the operation latency reported by the NFS client. While the evaluation results are encouraging, the relative performance results we report for NFS workloads are likely to be an underestimate. This is because our prototype was used only at the NFS server; the client counterpart of non-blocking writes was not engaged by this benchmark.

SPECsfs2008 operations are classified as *write, read,* and *others* which includes metadata operations such as *create, remove,* and *getattr*. For each variant solution, we report results for the above three classes of operations separately as well as the overall performance that represents the weighted average across all operations. Further, we evaluated performance when varying the relative proportion of NFS operations issued by the benchmark. The default configuration as specified in SPECsfs2008 is: reads (18%), writes (10%) and others (72%). We also evaluated three modified configurations: no-writes, no-reads, and one that uses: reads (10%), writes (18%), and others (72%) to examine a wider spectrum of behaviors.

We first perform a brief analysis of the workload to determine expected performance. Even for configurations that contained more writes than reads (e.g., 18% writes and 10% reads) the actual fraction of cache misses upon writes is far lower than the fraction of misses

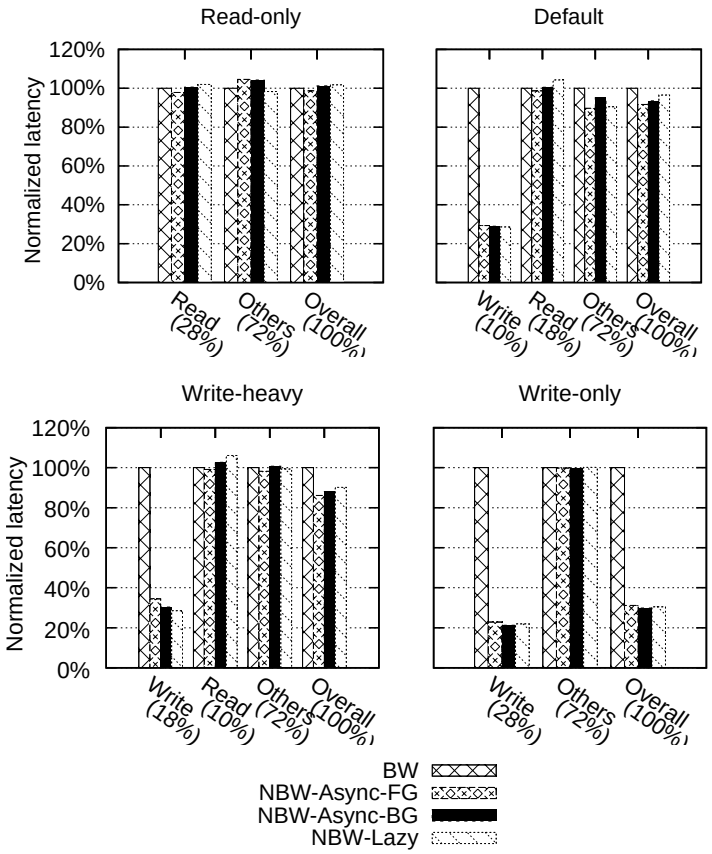


Figure 5.9: **Normalized average operation latencies for SPECsfs2008.**

due to reads (i.e., 16.9% write misses vs. 83.1% read misses). This mismatch is explained by noting that each read access to a non-cached page results in a read miss but the same is not true for write accesses when they are page-aligned. Further, Table 5.2 reports that only 39% of all writes issued by the SPECsfs2008 are partial page overwrites which may result in non-blocking writes. Thus, only 7% (i.e., 39% of 18%) of the operations the benchmark are writes that may result in write misses. As per Table 5.2, on average, reads are of 5 pages in size. If we compare 7% writes that could incur in a miss to only one page and 10% reads that could incur in misses to 5 pages, this translates approximately to 12.3% write misses vs 87.7% read misses.

Figure 5.9 presents the average operation latencies normalized using the latency with the BW solution. Excluding the read-only workload, the dominant trend is that the non-

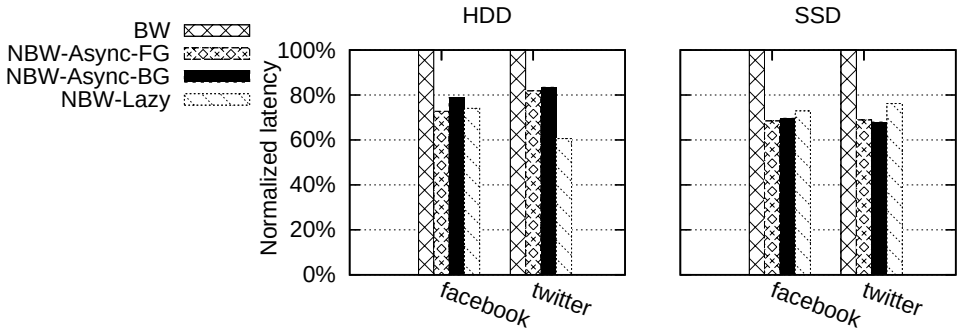


Figure 5.10: **Normalized average latencies when replaying MobiBench traces [ESO13].**

blocking write modes offer significant reductions in write operation latency with little or no degradation in read latencies. Further, the average overall operation latency is proportional to the fraction of write misses and to the latency improvements for NFS write operations. For the three configurations containing write operations, the latency of the write operations is reduced between 65 and 79 percent when using the different modes of non-blocking writes. Read latencies are slightly affected negatively due to additional blocking on certain pages. With BW, certain pages could have been fetched into memory by the time the read operation was issued. With non-blocking writes, the corresponding fetches could be delayed or not issued at all until the blocking read occurs. For the configuration with no write operations the average overall latency remained relatively unaffected.

5.5.3 MobiBench Trace Replay

The MobiBench suite of tools contains traces obtained from an Android device when using the Facebook and Twitter apps [ESO13]. We used MobiBench's timing-accurate replay tool to replay the traces. We fixed a bug in the replay tool prior to using it; the original replayer used a fixed set of flags when opening files regardless of the trace information. MobiBench reports the average file system call operation latency as the performance metric. We replayed the traces five times and report the average latency observed. Standard error of measurement was less than 4% of the average in all cases except one for which

it was 7.18%. Figure 5.10 presents the results for this evaluation for both hard disks and solid-state drives. Non-blocking writes exhibit a reduction in operation latencies between 20% and 40% depending on the mode and back-end storage used for both Facebook and Twitter traces.

5.6 Summary

For over four decades, operating systems have blocked processes for page fetch I/O when they write to non-cached file data. We revisited this well-established design and demonstrated that such blocking is not just unnecessary but also detrimental to performance. Non-blocking writes decouple the writing of data to a page from its presence in memory by buffering page updates elsewhere in OS memory. This decoupling is achieved with a self-contained operating system improvement seamless to the applications. We designed and implemented *asynchronous* and *lazy* page fetch modes that are worthwhile alternatives to blocking page fetch. *Lazy* page fetch mode is able to reduce memory consumption by avoiding complete-page allocations for sub-page size writes. We also designed *foreground* and *background* fetch mechanisms and implemented them within the *asynchronous* page fetch mode. Our evaluation of non-blocking writes using Filebench revealed throughput performance improvements of as much as 45.4X across various workload types relative to blocking writes. For the SPECsfs2008 benchmark, non-blocking writes reduced write operation latencies by as much as 65-79%. When replaying the MobiBench file system traces, non-blocking writes decreased average operation latency by 20-60%. Further, there is no loss of performance when workloads cannot benefit from non-blocking writes.

Non-blocking writes effectiveness can be limited by the amount of main memory available to buffer data while pages are still to be fetched from secondary storage. This amount of memory can vary greatly from time to time depending on the current workload being executed on the system. In the next chapter we present the design and implementation of new

memory tiering solutions that integrate emerging byte-addressable persistent memory into main-memory as an alternative to expensive DRAM. Our solution solves the non-trivial problem of dynamically migrating pages into appropriate tiers depending on page access frequency in order to boost system performance.

5.7 Credits

A preliminary design and evaluation of non-blocking writes by means of a simulation was published in the proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems in June 2011 [UKRV11] and was presented by Luis Useche. Luis Useche, Ricardo Koller, Raju Rangaswami, and Akshat Verma contributed the preliminary design of non-blocking writes. Luis Useche, Daniel Campello, Ricardo Koller, Raju Rangaswami, and Jesus Ramos substantially refined the preliminary design of non-blocking writes for its implementation in commodity operating systems. Luis Useche, Daniel Campello, Ricardo Koller, and Jesus Ramos implemented non-blocking writes with foreground asynchronous fetch mode for file system. Daniel Campello implemented background asynchronous fetch and lazy fetch modes for file system and refined the implementation to fully support SMP and kernel preemption. Daniel Campello and Hector Lopez executed all the experiments to evaluate the implementation of non-blocking writes to files. Daniel Campello created the tracing module used to capture the file system traces which motivated the use of non-blocking writes to files. This work was published in the proceedings of the USENIX Conference on File and Storage Technologies in February 2015 [CLU+15] and was presented by Daniel Campello.

CHAPTER 6

MANAGING TIERED MEMORY SYSTEMS WITH MULTI-CLOCK

In Chapter 5 we presented a solution that eliminates process blocking required to service file writes to pages that are not present in the OS cache in a way that is transparent to applications. Our non-blocking writes solution depends on the use of temporary main memory buffers to hold the data being written by applications. In this chapter we turn our efforts to make use of emerging byte-addressable persistent memory to augment the amount of available main memory for applications to use in a completely transparent way.

Over the last several decades DRAM performance and capacity have followed Moore's Law and thus kept up with advances in CPU technology. However, DRAM based memory systems have two significant drawbacks pertaining to cost and power consumption. These drawbacks impact their usage in both enterprise servers and mobile systems. While enterprises design and size their memory systems so as to keep workload working sets in main memory at all times, cost and power constraints also limit DRAM sizes within mobile systems. Today's multi- and many-core platforms support higher levels of workload parallelism (in mobile and server workloads) and multi-tenancy (in server workloads), placing greater demands on the memory system. Furthermore, the new generation of big data analytics applications in the enterprise and cloud are inherently memory intensive whereby workloads demand access to high-performance, yet low-cost, memory devices in preference to the much slower storage system [spa, sap, Fit04, Sal].

A complementary and disruptive technological change is the imminent availability of lower cost and lower power consuming byte-addressable persistent memory technologies [MSS12, Mit13]. These new memories offer an attractive set of properties that are well-suited to meet the growing memory demands of workloads at a fraction of DRAM cost (Table 6). They offer latency and bandwidth properties for byte-addressable access that are within an order of magnitude of those with DRAM. Furthermore, they do not

Parameter	DDR-DRAM	PM
Capacity per CPU	100s of GBs	Terabytes
Read Latency	1x	2x to 4x
Write Bandwidth	1x	1/8x to 1/4x
Estimated Cost	5x	1x
Endurance	10^{16}	10^{6} to 10^{8}

Table 6.1: **Comparison of Memory Technologies [DRZ+16].** Persistent memory (PM) characteristics are based on PCM and ReRAM technologies [Cro13, QSR09].

require power to retain data thereby greatly reducing the power consumption, which an alternate system with large quantities of DRAM would otherwise incur. Finally, when used to extend main memory, their persistence capability becomes irrelevant, thereby entirely avoiding the biggest performance overhead in using persistent memory [ZS15].

As with any emerging technology, we expect a gradual adoption of persistent memory over time, leading to initial systems where its use is completely transparent to legacy applications. One straightforward use of persistent memory is as a swap device with demand paging into DRAM, similar to how block devices are used. Paging is already supported in modern operating systems (OS), wherein persistent memory can be abstracted as a block device and then can be used as swap area. The main drawback of this approach, however, is that the natural byte-addressability of persistent memory is not utilized and any data in persistent memory must first be moved to DRAM before accessing it.

Another appealing usage of persistent memory is as a new tier in a multi-tier memory system with different tiers ordered from *high performance - low capacity* to *low performance - high capacity*. This approach allows applications to access their data directly from persistent memory without first paging in to DRAM. However, managing persistent memory simply as additional available memory could compromise the effectiveness of the tiered memory system. Once an application has exhausted higher memory tier resources, future allocations for said application or any other application on the system will have to be ser-

viced from lower tiers. Additionally, such allocations will remain in lower tier memory regardless of how important (hot) the data is until they are freed.

We make the following contributions in this chapter:

1. We identify and study alternate approaches to including byte-addressable persistent memory in today's memory systems: demand paging, static tiering, dynamic tiering.
2. We propose MULTI-CLOCK, an OS-resident page management mechanism that enables dynamic, timely migration of hot data to the higher memory tier and cold data to the lower tier with the objective of optimizing application performance. MULTI-CLOCK uses both DRAM and persistent memory in a complementary manner and can benefit from persistent memory's high capacity while still retaining DRAM's low latency for frequently accessed pages.
3. We implement a prototype for MULTI-CLOCK using Linux version 4.0 by extending the kernel's memory zones to represent different memory tiers and extending its page reclamation algorithm to include dynamic tier migration logic.
4. We evaluate the performance of our prototype using workloads that include micro-benchmarks to graph analytics, KV, and database benchmark workloads.

6.1 Motivation

Persistent memory forces a rethink of how workloads consume both memory and storage. The memory usage is relatively more straightforward given that persistent memory devices provide byte-addressability. Furthermore, this usage is rather appealing given the high cost and power demands imposed by DRAM technology. We estimate the cost of DRAM to be 5x of persistent memory for the same capacity [DRZ+16].

Today, enterprises are hosting applications with large working sets. Many of these workloads are memory intensive, requiring most of their working set to be readily avail-

able [spa, sap, Fit04, Sal]. Mobile systems are also running increasingly memory intensive applications [chr]. Thus, we anticipate both ends of the market, enterprise and mobile, to drive demand for the new memory technology. However, using these new persistent memory most effectively is non-trivial.

6.1.1 Swapping vs. Tiering

A straightforward approach to integrating persistent memory is to use it as a swap device within a demand paged memory architecture. Persistent memory is attractive as a paging device due to its higher performance when compared to conventional block-based storage devices. Demand paging works by storing and retrieving data to and from a swap area (conventionally a block-based storage device) at the page granularity. Today operating systems already support this use. Linux, for instance, exports a block abstraction of available persistent memory in the system via the `/dev/pmem` device which can be used as a swap via `mkswap` and `swapon` commands. This block abstraction has been used in the past to enable the use of persistent memory to store file system data [DAX, DKK+14].

An alternative approach is to configure both DRAM and persistent memory as separate tiers in a multi-tier memory system. With tiering, data residing in the byte-addressable persistent memory is treated as resident in main memory and is readily accessible for the CPU to use. Tiers are defined by disjoint physical memory regions or, in other words, disjoint sets of memory frames. On start-up, the operating system identifies which frames belong to each memory type and assigns them to their proper tier. Tiers then get arranged in a specific order, following the characteristics of the different types of memory (i.e., from *high performance - low capacity* to *low performance - high capacity*), to service memory allocations. There is no explicit limit on the number of different tiers that can be defined. Assuming temporal locality of reference, allocations are served from the highest performing tier to boost application performance.

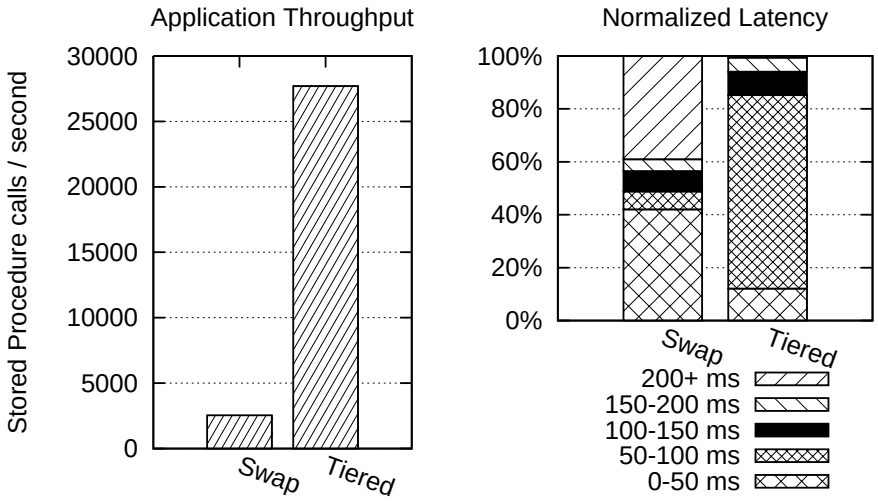


Figure 6.1: **TPC-C performance using persistent memory as swap vs as a new tier.** The left graph shows application throughput reported by the benchmark while the right graph presents the distribution of latency for each benchmark operation executed.

To compare the difference in performance of both approaches we ran the on-line transaction processing benchmark, TPC-C [Tra], on top of the VoltDB [Vol] database (see section 6.4.5 for more details on the configuration used). We executed the benchmark on (1) a system configured to use persistent memory as a block swap device and (2) a system with two tiers of memory: DRAM and persistent memory. Figure 6.1 presents a comparison between these two approaches.

The left graph on Figure 6.1 shows how persistent memory, when used as a swap, results in poor application performance, about 9% of the performance achieved when used as an additional memory tier. The performance gap can be explained by the need of the paging solution to first move data from persistent memory into DRAM before the CPU can access it. This memory movement overhead is reflected in the latency of the benchmark operations reported in the right graph of Figure 6.1. Additionally, with demand paging there is a need to maintain several data structures to manage the swap area contents.

6.1.2 Static Tiering

Tiering as described earlier, falls into the category of static tiering, since allocations once mapped to a tier may not get reassigned to a different tier. Static tiering, however, is not a complete solution since it does not guarantee fairness in memory access performance in a multiprocessing system. If an application wins the race to allocate memory from a higher tier and such space in exhausted, future allocations will be downgraded to use lower tiers during their entire lifetime, regardless of how the *importance* of the contained data changes over time. We define importance of a data page as how frequently the page is accessed in the near past, or in other words, how hot the page is with respect to the other pages in the system. To understand the significance of this problem, we designed a simple experiment involving two processes. Both processes execute the same benchmark (TPC-C over VoltDB) as in the previous experiment. *Process 1* gets to execute first and it uses up most or all available memory in the higher tier (DRAM). Figure 6.2 shows how performance for *Process 2* degrades when it may only use the lower tier (persistent memory) to service most or all of its allocations. In this experiment each process consumes about 34GB and we vary the amount of available DRAM in the system and the latency of accessing persistent memory following the range presented in table 6. We see how the performance of *Process 2* degrades on relation to the performance of *Process 1* as there is more available DRAM for *Process 1* to use and not *Process 2*. The problem becomes more acute as the latency gap between tiers is increased. Section 6.4 describes more details on this experiment.

6.1.3 Dynamic Tiering

Problems with static tiering can be overcome with a solution that dynamically migrates important pages to higher tiers and less important pages to lower tiers. MULTI-CLOCK builds on this idea and addresses several key challenges involved in creating a usable solu-

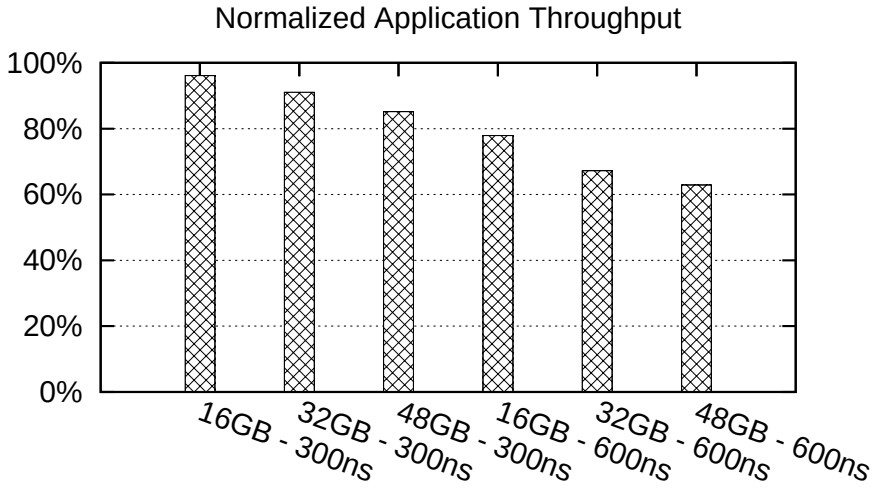


Figure 6.2: **Relative TPC-C performance of two instances executed in succession.** Performance of the second instance relative to the first. Varying in the x-axis, amount of available DRAM in the system and accesses latency to emulated persistent memory.

tion. First, a migration mechanism that allows the contents of a page be moved from one physical frame belonging to a particular tier to a frame in a different tier becomes necessary. Such a mechanism must also fix any virtual→physical address mapping that could exist on the system at any time (i.e., page table entries, indexes to file-backed pages in the page cache). Second, with dynamic tiering the system must adapt to workload changes in a timely fashion, in other words, it must decide when is the right moment to trigger memory migrations. The next section presents how MULTI-CLOCK selects candidates to be migrated between memory tiers and decides when to carry such migrations in a manner that is transparent to applications running in the system.

6.2 MULTI-CLOCK

A fundamental problem with static tiering is the mismatch of page access performance requirements with tier performance capabilities. MULTI-CLOCK addresses this problem with a solution that dynamically migrates important pages to higher tiers and less important

pages to lower tiers. To achieve this goal, MULTI-CLOCK applies a modified version of the Page Frame Reclamation Algorithm (PFRA) used in Linux (which is based on the CLOCK algorithm) to each memory tier separately. The CLOCK algorithm approximates LRU by checking for references when scanning the list of pages and moving any referenced page to the head of the list. If at any time a tier cannot be used to service an allocation, it is marked as being under memory pressure. Once a tier is marked as being under memory pressure, MULTI-CLOCK frees up space in that tier by picking candidate pages to be migrated to its lower tier, if one exists, or evicted out of memory altogether. Every page in the system is arranged in one of its tier's three lists according to their degree of hotness/coldness in terms of accesses. We call these lists the *inactive, active* and *promote* LRU-like lists. The top tier in the system does not use a promote list since there is no higher tier to migrate pages to. For systems with only one memory tier, MULTI-CLOCK trivially reverts to the original PFRA behavior since it only uses *inactive* and *active* lists.

6.2.1 Life Cycle of a Page

Every list is scanned at various points in time to make decisions regarding migrations. A recently allocated page starts in the inactive list which maintains candidate pages for *demotion*, i.e., migration to a lower tier. A page is said to be referenced if any type of access (i.e., read or write) occurs to such page. Both inactive and active lists make a differentiation between pages referenced and not referenced since the last scan.

During a scan, if a page has been marked referenced since the previous scan, it is then marked as not referenced and skipped over, as CLOCK does. On the other hand, if the page was not referenced, it is moved according to which list belongs to: (a) if it belongs to the active list it is moved to the inactive list, (b) if it belongs to the inactive list is then migrated to its lower tier, and if none exists, evicted out of memory. This movement of pages out of a list is referred as the shrink of the source list. At the same time, when an access occurs

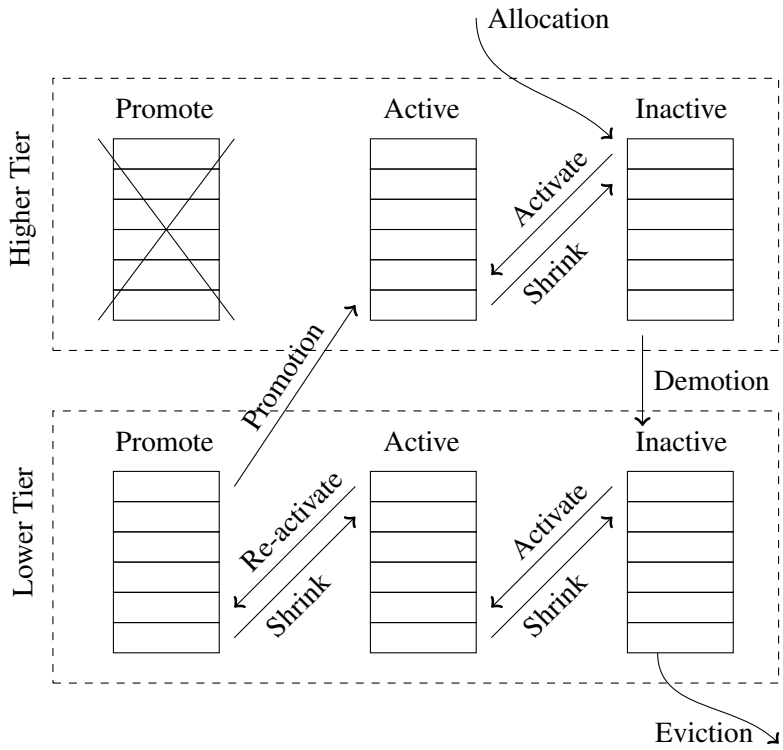


Figure 6.3: **MULTI-CLOCK design.** The system consists of two memory tiers, each with inactive, active and promote LRU-like lists. Arrows represent the possible movement of pages between lists and tiers. This design generalizes to more than 2 tiers.

to a page on the inactive list and that page was marked as referenced, this page is *activated* by being moved to the active list where it starts out by being marked as not referenced. A similar mechanism is followed when a page is *re-activated* and is moved from the active to the promote list, where it becomes a candidate for *promotion* (i.e., migration to an upper tier).

Figure 6.3 depicts this overall arrangement of lists in the two tiers on the system and the possible movement of pages within and across these tiers. With this arrangement, the system is able to classify pages into three categories: hot, warm and cold. Hot pages navigate the lists within a tier and eventually reach the promote list were they become candidate pages to migrate to the upper tier. On the other extreme, cold pages remain in the inactive list where they become candidates for migration to a lower tier when the tier

experiences memory pressure. The system is then able to make decisions on how to adapt and place each page in their proper tier according to their frequency of accesses.

A key challenge is keeping track of accesses and updating the reference status of pages in a timely matter. This is addressed differently depending on the type of page accesses used by applications. Applications can access memory pages in two ways: *unsupervised*, by memory mapping pages into their address-space, and *supervised*, using the operating system's (OS) file system call interface.

Supervised Access

This type of access is typically used for file backed pages and it gives the OS control at the moment of the access to perform the necessary book-keeping. When applications use supervised access to memory pages, the operating system has control to mark these pages referenced (for e.g., in Linux, via `mark_page_accessed()`) and, if necessary, to move between lists (activate or re-activate) before even processing the requested data access.

Unsupervised Access

Accesses to anonymous or file-backed memory that is directly mapped into the application's virtual address space via `mmap` are more difficult to monitor. This type of access is entirely unsupervised and the OS is not able to mark such pages. To handle unsupervised access, MULTI-CLOCK relies on the page referenced bit set by the CPU in the process' page table entry. During each scan as described earlier, before making any decision regarding a particular page, MULTI-CLOCK checks within every process' page table that maps it for a referenced bit set. If a referenced bit is found set, MULTI-CLOCK updates the page status and takes care of the necessary movement between lists (i.e., mark as referenced, activate, or re-activate the page).

6.2.2 Promotion Mechanism

Periodically, a new system daemon, *kpromoted*, is woken up to scan the lists, updating them, and to migrate any pages from the promote list to a higher tier due to recent unsupervised accesses. Implicitly, MULTI-CLOCK relies on the periodicity of *kpromoted* waking up to ensure that hot pages in lower tiers are migrated to higher tiers in a timely manner. The frequency of *kpromoted* execution defines the capacity of the system to react quickly to workload changes. On the other hand, if scheduled too frequently, excessive context switches to accommodate its execution could also affect application performance. Careful tuning of *kpromoted*'s execution schedule is necessary to ensure that applications benefit from the promotion mechanism in MULTI-CLOCK. In the prototype system we built, we chose the kpromoted execution schedule to be every 100ms and this worked fairly well for the workloads we evaluated the system with. It resulted in sufficient responsiveness in promoting hot pages without imposing high CPU overheads due to unnecessary scanning of every page in the LRU-like lists.

6.2.3 Demotion Mechanism

Demotion allows moving cold pages from a higher tier to a lower tier when they are no longer sufficiently important. MULTI-CLOCK's design of this mechanism is based on page eviction design in today's systems. To avoid running out of memory on a given tier, a tier is marked under memory pressure proactively when it reaches specific watermark levels. These levels are calculated by the system according to the amount of memory in the tier vs the total amount of memory in the system.

If any tier is marked as being under memory pressure, each list is scanned with the objective of freeing up memory. Any page in the promote list is first attempted to be migrated to a higher tier and if that is not possible (for instance, the page is locked), it is

moved to the active list. If the higher tier is also under memory pressure, promotions from the lower tier result in immediate page demotions from the higher tier. Next, if the ratio of pages in the active list with respect to the inactive list exceeds a tunable threshold (inherited from PFRA and typically $\sqrt{10 * n} : 1$, where n is the amount of memory in GB available in the tier), pages not marked as referenced in the active list are moved to the inactive list. Finally, the inactive list is scanned in search of pages not marked as referenced to be migrated to a lower tier. Migration might not be possible, specifically because the candidate pages belong to the lowest tier in the system. In this case, these pages are written back to block storage (i.e., file-backed pages to file system and anonymous pages to the swap area if available) before triggering the out-of-memory (OOM) killer as the last option.

6.3 Implementation

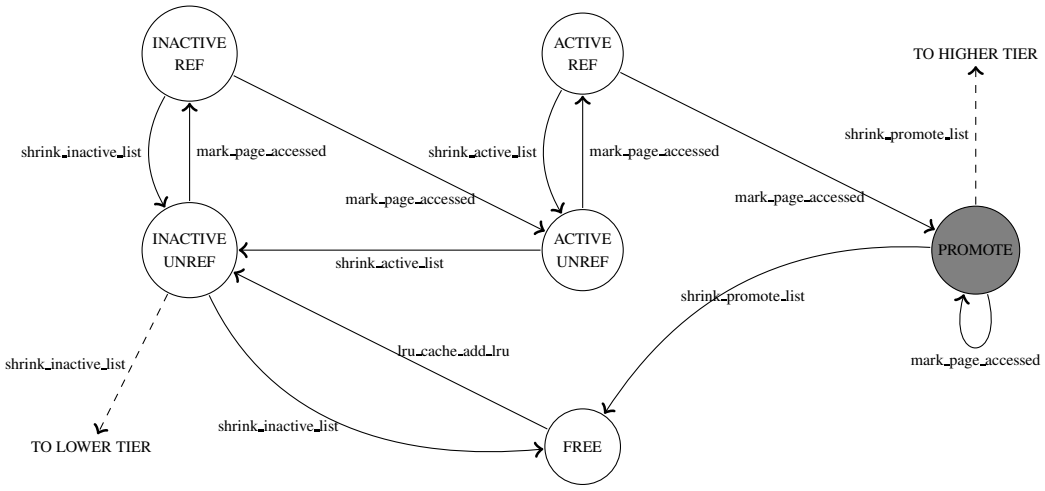


Figure 6.4: **Page state diagram depicting the Linux implementation of MULTI-CLOCK** Each vertex represents a page state; white vertices are original PFRA page states while the shaded vertex is a new page state introduced by MULTI-CLOCK. Solid edges represent Linux procedures that change page state; dashed edges represent page migration to a different tier. Counterparts to shrink_list methods are implicit on page allocations which cause lists to expand.

The existing Linux mechanism to describe physical memory relies on the definition of *nodes*. In NUMA architectures, each bank of memory is represented by a single NUMA node. On the other hand, for UMA architectures, Linux uses a single NUMA node to represent all physical memory in the system. The data structure used to represent nodes is called `pglist_data`. Each node is then divided into memory ranges called *memory zones* and Linux uses the data structure `zone` to represent them in memory. Zones are of different types, and each type is suitable for different usages (i.e., `ZONE_DMA` gathers physical addresses that can be accessed by legacy hardware through DMA).

We implemented a prototype of MULTI-CLOCK in the Linux kernel v4.0. We used the concept of memory zones to define new memory tiers. Our prototype evaluates a two tiered memory system: one tier of DRAM and another of persistent memory; the latter is emulated using the Intel HMEP platform (discussed in Section 6.4.1). The DRAM tier is defined by the existing Linux memory zones; we defined a new memory zone, `ZONE_PMEM`, that encapsulates all physical persistent memory. DRAM memory zones are scanned when under memory pressure to find demotion candidates to be migrated to `ZONE_PMEM`; `ZONE_PMEM` scans for demotion candidates may result in evictions from main memory to secondary storage if `ZONE_PMEM` becomes full at any point in time. We rely on the existing Linux migration mechanisms already in place for the hot-plug/hot-remove of memory. Linux's page migration mechanism (`migrate_pages()`) is in charge of allocating new memory pages given an allocation routine, copying the memory contents from origin pages to newly allocated destination pages, and fixing any memory mappings that refer to the migrated pages.

Originally, each Linux memory zone maintains their own set of LRU-like lists: anonymous inactive, anonymous active, file inactive, file active, and unevictable. We added two lists: *anonymous promote* and *file promote*. Such promote lists are only in use for `ZONE_PMEM`. Unevictable pages belong to the unevictable list and are such pages in the

system that are locked into memory (typically via `mlock()`) and cannot be evicted nor migrated. Every evictable page in the system, depending on being file-backed or anonymous, will belong to one set of LRU-like lists (anonymous lists or file lists) and it will traverse these by transitioning through different states as depicted in Figure 6.4. We also extended the `struct page` flags which maintain the status of a page during its existence to add a new flag: `PagePromote`. This new flag is used by the OS to mark that the page in question, which is to be added to the zone's lists, belongs to the promote list. The memory overhead of these modifications is negligible since we reused the list pointer on the `struct page` to index the pages in the promote lists; we also reused the space allocated for the page flags to maintain the newly defined flag.

We implemented the system daemon discussed in Section 6.2.2 as a new kernel thread, *kpromoted*, which is woken up every 100ns to carry the migration of any pages sitting in the promote list to a higher tier. This thread's design follows those of PFRA for the *kswapd* eviction daemon: one kernel thread per NUMA node. This design aims to avoid lock contention on critical per-node data structures. Maybe the most relevant data structure that each NUMA node has is its own version of the memory zones in the system according to the amount of memory of such zones that is physically local to the node in question. Each zone also contains its own version of the LRU-like lists described in the previous paragraph to maintain the status of the zone's pages. This in turn means that since each NUMA node protects the accesses to the zone structures through the `zone->lock`, more kernel threads scanning memory pages could degrade under heavy lock contention to access the LRU-like lists.

Our implementation of the MULTI-CLOCK algorithm is encapsulated mostly in `mm/vmscan.c` and `mm/swap.c`. Table 6.2 presents how much new code was added for MULTI-CLOCK and which files were modified in the Linux's source code on top of an existing static tiering implementation. We extended `mark_page_accessed()` to check

Source File	New Lines	Modified Lines
include/linux/gfp.h	0	1
include/linux/mmzone.h	14	4
include/linux/page-flags.h	3	1
mm/debug.c	1	0
mm/memcontrol.c	3	1
mm/page_alloc.c	1	2
mm/swap.c	16	3
mm/vmscan.c	154	5
mm/vmstat.c	4	0

Table 6.2: **Linux source code modifications in number of lines.**

for pages that are already referenced and marked as active and are being referenced again to mark such pages as promote page with the `PagePromote` flag and to move them from their correspondent active list to the promote list (see transition from ACTIVE REF to PROMOTE in Figure 6.4). We created a new `shrink_promote_list()` method that complement the existing `shrink_active_list()` and `shrink_inactive_list()` methods to handle movements of pages out of the promote list. Migrations to the upper tier are handled via `shrink_promote_list()` and migration to the lower tier (or evictions) are handled via `shrink_inactive_list()`. Both methods result in a physical frame in the tier being freed after successful migration of its contents.

6.4 Evaluation

We address the following questions:

(1) What is the overhead experienced by applications due to page migration between tiers?

(2) How do applications using MULTI-CLOCK perform relative to using static tiering?

(3) How sensitive is MULTI-CLOCK to the performance of the underlying memory technology?

(4) How well does MULTI-CLOCK handle workload changes?

(5) What other overheads do MULTI-CLOCK mechanisms impose on the system?

We use a micro-benchmark to address (1). We use GraphLab's PageRank implementation over the Twitter dataset to answer question (2). We use YCSB benchmark's platform and workloads over Memcached key-value store back-end to answer (2), (4) and (5). Finally we use VoltDB and the TPC-C benchmark to further analyze questions (2) and (4) and address question (3). The MULTI-CLOCK prototyping platform runs Ubuntu Linux 14.04.4 LTS (kernel 4.0)

6.4.1 Emulation Platform

For evaluating MULTI-CLOCK, we used the hybrid memory emulation platform (HMEP) [MDS+15] developed by Intel. HMEP uses custom CPU microcode to configure latency and bandwidth for a separate range of physical memory to emulate persistent memory with a portion of the DRAM. HMEP is tied to Intel Xeon E5-4620 platforms, with 16 2.6 GHz cores distributed between two processors. The observed DRAM latency on the system is 150ns. For our experiments, we configured HMEP to set the latency of accesses to the emulated persistent memory at 300ns to simulate fast PM devices and 600ns for relatively slower ones. This experiment setup has been used in previous work on tiered memory [DRZ+16].

6.4.2 Micro-benchmark

To obtain a sense of the overhead involved in continuously migrating an application's pages between available tiers we wrote a small micro-benchmark. We also added to the kernel, a `/proc` hook where an application can request its own address space to be migrated to a target memory tier. Writing to this hook is a blocking operation that does not return until all

the memory pages belonging to the application's address space in question are completely migrated to the target tier.

This simple micro-benchmark works as follows: (a) it allocates a target page, (b) spawns a number of threads with the only purpose of continuously calculate a check-sum on the contents of the shared target page, and (c) the parent thread continuously triggers migration of all the application's memory pages back and forth between two distinct memory tiers. Migration is continuously triggered by simply writing over and over again to the `/proc` hook as soon as the previous write completes.

Linux's page migration works by taking a list of pages and migrating one page at a time. By having multiple threads accessing the same page and having the application allocate a small amount of memory, we create a scenario where we can expect the target page to be under migration most of the time. While a page is being migrated, the application's page table is fixed to remove any mapping to the old location of the page; the page is `memcpy`'ed to the new location and then the mapping in the page table is updated. Application accesses during page migration will incur page faults, stalling execution and thus negatively impacting performance. The above scenario aims to present a worst-case performance behavior of the page migration mechanism which is at the heart of MULTI-CLOCK.

Figure 6.5 presents the results obtained from our micro-benchmark evaluation. We measured the amount of time invested in page fault handling for the 2 configurations being evaluated: with and without continuous page migrations. As expected, the time increases from virtually nothing for the case when no migration occurs to about 2250 milliseconds for the case of continuous migration. This amount of time spent on page fault handling represents 1.25% of the 180 seconds of total execution time. This result is then consistent with the performance results yielded by the micro-benchmark shown in Figure 6.5. Total transactions achieved by the micro-benchmark suffer a degradation of just 3.3% when continuous migration triggered between memory tiers is compared to no migration at all. From

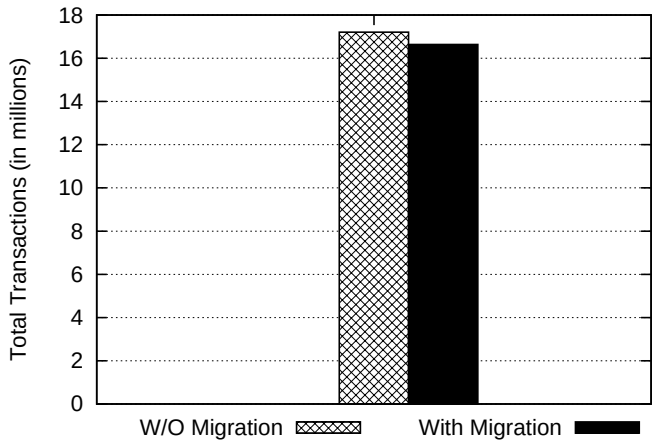


Figure 6.5: **Micro-benchmark performance with and without continuous migrations.** The benchmark runs for 180 seconds with 9 worker threads. On Y axis total number of transactions completed (check-sums on target page)

this experiment we can conclude that the page migration overhead is almost negligible to application performance and the expected benefits of proper placement of pages to memory tiers is likely to outweigh any migration overhead.

6.4.3 GraphLab

We address question (2) using GraphLab, a large scale memory intensive application. GraphLab is a framework for graph analytics capable of running a wide variety of graph processing algorithms. One such algorithm is the PageRank which we used for this experiment on the 25GB Twitter dataset [KLPM10].

The execution of PageRank is divided in two stages: a load phase and an execution phase. In the load phase, the input file containing the graph data is read into memory and the in-memory version of the graph is constructed. During the execution phase, the actual PageRank algorithm is executed over the already memory resident graph representation of the data. The execution of GraphLab's PageRank on the Twitter dataset during the execution phase consumes about 96GB of physical memory (resident memory) and 126GB of virtual memory (including shared memory and files mapped).

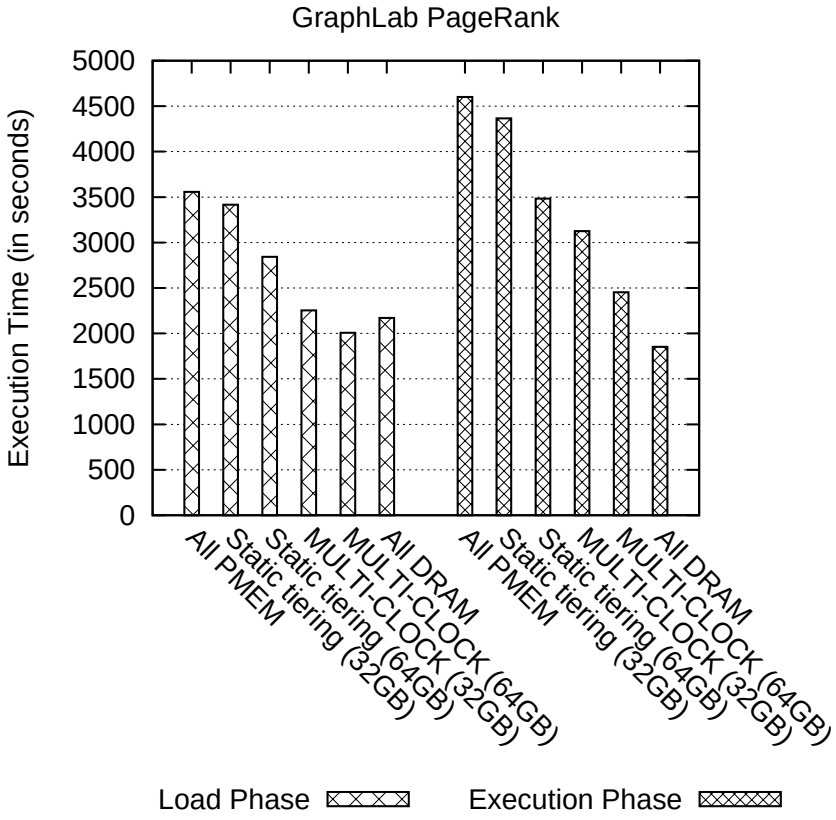


Figure 6.6: **GraphLab's execution time of PageRank on the Twitter dataset.** Y axis present execution time (lower is better). Left cluster of bars present execution time for the load phase of the benchmark while right cluster present results for the execution phase. For each cluster, left and right extremes bars represent baseline results: on the left execution using only persistent memory, on the right using only DRAM. Results in between compare static tiering vs MULTI-CLOCK while varying the amount of total DRAM in the system from 32GB to 64GB.

Figure 6.6 presents the results of executing GraphLab's PageRank under 6 different system configurations. The graph present bars divided into two clusters: the left cluster with results for GraphLab's load phase and the right cluster with results for the execution phase. On the far left of each cluster we present the results of executing the application only using persistent memory as the worst case result. On the far right, we have the baseline results of executing the benchmark on a system with enough DRAM to hold the complete working set. As expected, for both static tiering solution and MULTI-CLOCK, as more

DRAM is available in the system, more of the application's working set can be hosted in the higher performing tier, reducing data access latency and thus reducing execution time.

GraphLab's load phase is rich in new allocations wherein the MULTI-CLOCK algorithm makes sure that cold data is demoted from the higher tier in a timely fashion to allow new allocations to be serviced from DRAM. The load phase performance with MULTI-CLOCK is within 4% of the all DRAM configuration. However, the two variants of static tiering incur 30% and 60% degradation. On the other hand, GraphLab's execution phase is not expected to perform as many allocations as the load phase; instead it is expected to access all allocated memory representing the in-memory graph in order to execute the PageRank algorithm. This phase takes advantage of both MULTI-CLOCK's demotion and promotion algorithms to ensure that the hottest data is readily accessible in the high-performing tier while the mostly unused cold data is relegated to the lower, low-performing tier.

In the execution phase of the experiment, the benchmark with MULTI-CLOCK required only 53% and 68% of the time required for the all-persistent memory configuration when 64GB and 32GB of DRAM were available respectively. In comparison, static tiering configurations require as much as 75% and 95% of the execution time required by the all persistent memory solution. The benchmark speedup of only 5% with static tiering when only 32GB DRAM were available rendered this amount of expensive DRAM almost useless in improving the all persistent memory baseline performance and not worth the cost. Static tiering presents 88% and 135% degradation on the execution phase performance when compared to the DRAM baseline. In comparison, MULTI-CLOCK's performance is only 32% worse compared to the DRAM baseline when 64GB of DRAM were available and 69% when just 32GB of DRAM was available.

Figure 6.7 presents the same results of Figure 6.6, now taking into consideration the expected cost per GB of the different tiers. Here we can observe the cost reduction benefits of MULTI-CLOCK when more persistent memory is used in place of DRAM. A similar cost

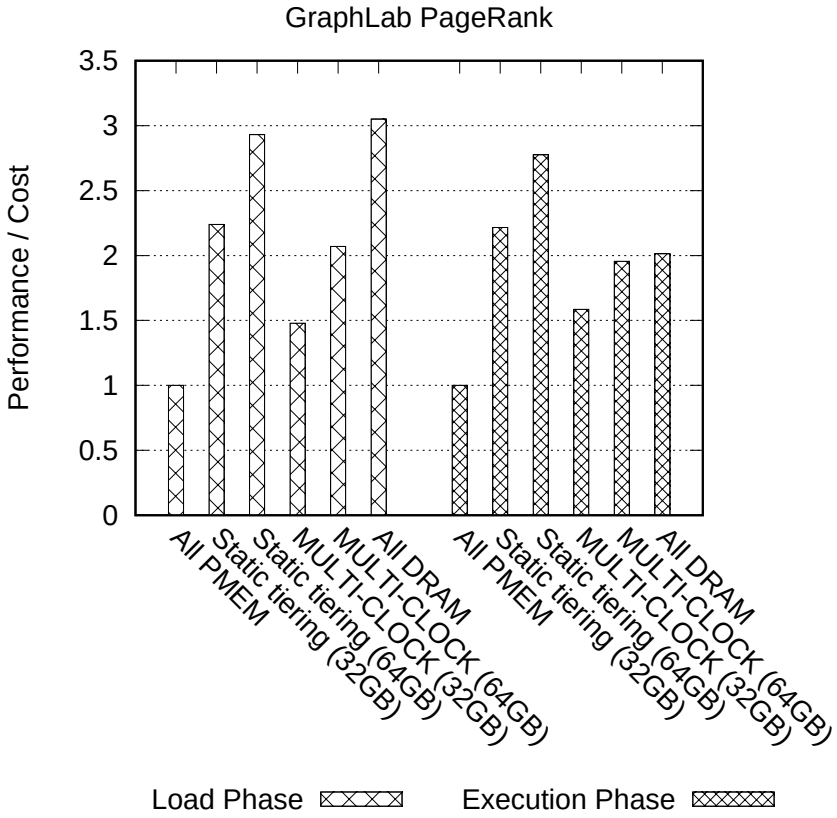


Figure 6.7: **Time*Cost comparison for GraphLab's PageRank on the Twitter dataset.** Y axis present Time*Cost (lower is better). DRAM's cost is 5x cost of persistent memory.

study can be used on a case-by-case basis to size the different tiers in the multi-tier system appropriately depending on the workloads to which the system is going to be exposed.

6.4.4 Memcached YCSB benchmark

We use the Yahoo! Cloud System Benchmark (YCSB) [CST+10] to answer questions (2), (4) and (5). YCSB is a framework for evaluating different key-value stores and includes a set of core workloads identified as A through F that can be executed by their workload generator. This generator is written as a client application which communicates with diverse back-end servers. For our evaluation we used Memcached [Fit04], an in-memory cache service which uses a large amount of main memory to maintain its data, as the key-value

store back-end of YCSB. Memcached also represents a real-world application that is being widely used in the industry today.

YCSB workloads are divided in two phases: a load phase and an execution phase. The load phase is in charge of populating the back-end key-value store with the required number of records, which in our evaluation is set to 60 million. With 60 million records, Memcached consumes about 83 GB of main memory. On the other hand, the execution phase carries diverse types of operations over the back-end depending on the description of each workload. One thing to note is that YCSB's workload E makes use of SCAN operations that may or may not being implemented by the different key-value back-ends. Memcached does not implement SCAN operations, making workload E non-operational. Further, the load phase is the same for all workloads and most workloads (all but D and E) do not change the amount of records in the back-end. In order to gather the results of this experiment we followed the prescribed execution sequence [ycs] for the YCSB workloads. Since workload D changes the amount of records in the back-end, the order of execution is arranged in the following manner: Load Phase, Workload A, Workload B, Workload C, Workload F and Workload D. For this evaluation we set our emulation platform to have 32 GB of DRAM and 192 GB of persistent memory with 600ns latency for accesses.

Performance comparison

We first answer question (2). Figure 6.8 presents the results of our evaluation with YCSB against Memcached as the key-value store back-end. We can observe how MULTI-CLOCK's placement of frequently accessed pages in DRAM translates to application throughput benefits of up to 10% when compared to static tiering depending on the type of workload. Figure 6.8 also presents the results obtained from running the benchmark on a system with only DRAM as the upper bound for performance. The improvement in performance achieved by MULTI-CLOCK for some workloads can be really close to this upper bound (Workload

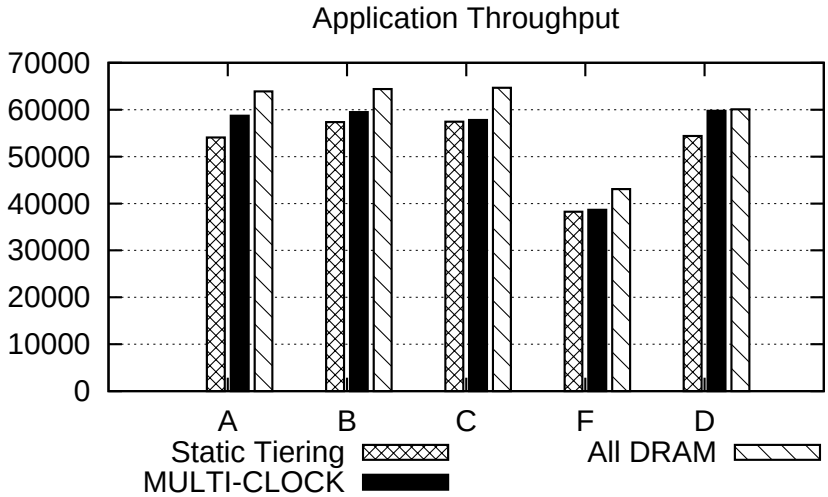


Figure 6.8: **YCSB throughput of different workloads running against Memcached.**

D, where new records are inserted, and the most recently inserted records are the most popular) showcasing the benefits of automatic intelligent migration.

Performance analysis

To better understand how MULTI-CLOCK benefits are reflected in the different workload executions, we answer questions (4) and (5). Figure 6.9 presents two sets of statistics regarding MULTI-CLOCK background activity over time. On the top graph of Figure 6.9 we have the CPU utilization of the two daemons in charge of page migrations: kswapd and kpromoted. The first section of the graph depicts the activity of the system during the Load phase of the benchmarks. This phase is characterized by many new memory allocations required for the creation of new records in the key-value store, which in consequence generates heavy memory pressure on the system's highest memory tier. This memory pressure translates to an average of 15% CPU utilization for the kswapd daemon which handles page demotions. In contrast, during the different workload executions, since memory allocations are not as frequent or none existent altogether, the amount of CPU utilization for kswapd is reduced by half, to about 7.25% in average. The demotion activity is still present when

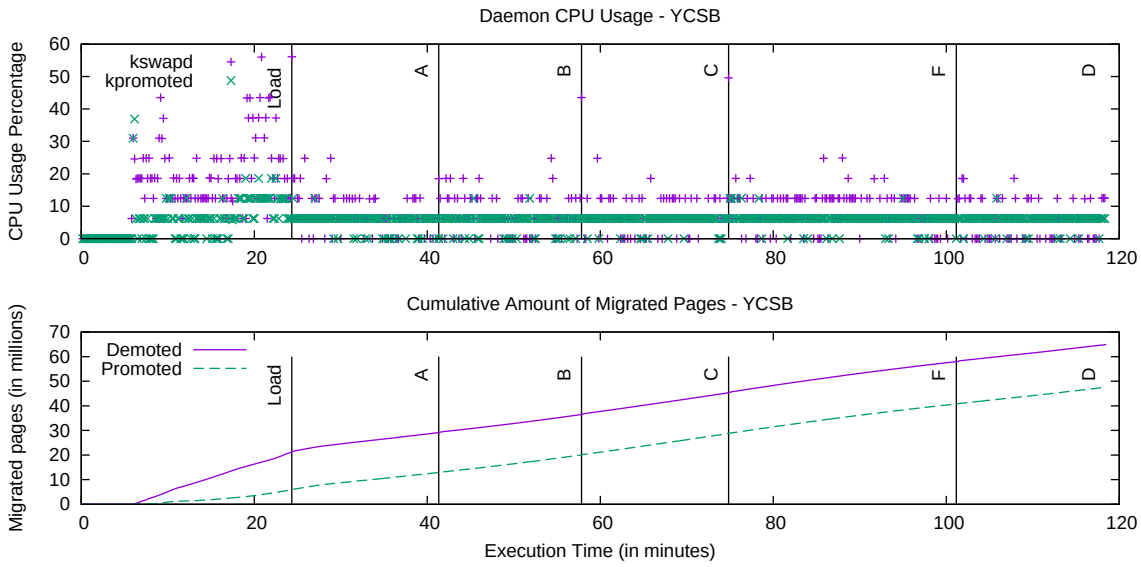


Figure 6.9: **MULTI-CLOCK statistics during YCSB benchmark execution.** On the top graph: CPU utilization for kswapd and kpromoted kernel threads over time. On the bottom: Cumulative amount of pages demoted from DRAM and promoted from persistent memory over time.

no new allocations are made by the application in order to make room in the higher tiers for pages being promoted from lower tiers. kpromoted's CPU utilization is limited to about 5.5% on average. We note that kswapd's CPU utilization is typically higher than kpromoted's since it has to keep up demoting pages to free up space in response of promotions and new memory allocations as well.

On the bottom plot of Figure 6.9, we report the number of pages being migrated across tiers on the system. Two key observations are derived from this graph. The first observation is that during the load phase, the rate at which demotions occur is higher than any other point in time, while in contrast, promotion rate is really low. This behavior is an outcome of the memory pressure that arises at the high-performing tier of the system when new memory allocations are requested by applications. We note that some promotions actually occur during the load phase and these may be related to how the back-end manages its own memory pages to service internal allocations of sizes smaller than a page. The second observation is that during the execution phase of the different workloads the rate of demotions

is roughly similar to the rate at which promotions occur. This is because demotions become necessary to make room for pages being promoted from lower tiers and not as much due to new memory allocations.

6.4.5 VoltDB TPC-C benchmark

We used the industry standard TPC-C benchmark on top of VoltDB [Tra, Vol] to evaluate MULTI-CLOCK and analyze questions (2), (3) and (4). VoltDB is an ACID-compliant, distributed, in-memory relational database. The TPC-C benchmark imitates the operations of a wholesale product supplier. The number of warehouses was set to 512 and the number of sites in VoltDB was set to 8. The benchmark reports TPC-C throughput measured as total transactions per second.

The core benefit of MULTI-CLOCK, we believe, is its ability to adapt to workload changes in the system. To evaluate this property, we first run one instance of the VoltDB, execute the TPC-C benchmark, and note the reported throughput. This instance is configured to consume around 34GB of main memory. Depending on the amount of DRAM in the system, these allocations may or may not be serviced completely by the high performance DRAM tier. Next, in the second phase of the experiment, we execute a second, identical VoltDB/ TPC-C benchmark instance (using 34GB of memory as well), without allowing the first VoltDB instance to free any used memory. In this second phase, memory allocated by the first VoltDB instance is unused and thus cold.

Figure 6.10 contrasts the results from a static tiering solution against dynamic tiering using MULTI-CLOCK. Each data point represents the relative throughput of the second VoltDB instance with respect of the first one. On the X axis we vary the amount of available DRAM and the emulated latency for persistent memory. In each case, the amount of persistent memory was set to 192GB.

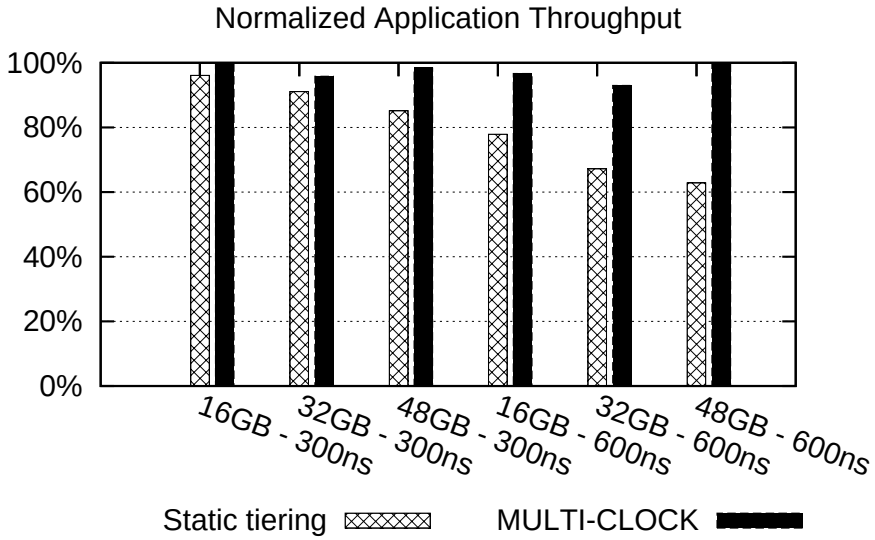


Figure 6.10: **TPC-C benchmark performance of two instances executed in succession.** Performance of the second instance relative to the first while varying amount of DRAM and accesses latency to emulated persistent memory.

With static tiering, the second VoltDB instance is forced to make allocations from the low performance persistent memory due to the first instance exhausting all available DRAM. While the amount of DRAM increases, the gap in performance of both instances increases too, given that the first instance is able to fit more allocation in fast DRAM and exhibit greater performance. Additionally, when the latency of accesses to persistent memory increases, the performance degradation of the second instance increases too. In contrast, when the second instance of VoltDB executes the TPC-C benchmark on MULTI-CLOCK, the system is able to identify all cold pages belonging to the first VoltDB instance and demote them to the slower tier. New allocations are then serviced from fast DRAM and the reported throughput for the second instance is almost equivalent to the first instance.

6.5 Discussion

In our prototype implementation of MULTI-CLOCK we used separate Linux memory zones to represent the physical memory of each memory tier containing different memory tech-

nologies. An alternate approach to represent each tier is to use NUMA nodes instead. Each NUMA node also gathers separate sets of physical memory in the system and there exist tools (e.g., `numactl`, `migratepages`) to statically control how an application's memory allocations are serviced and migrated from available nodes in the system. Linux memory zones have been used to describe different capabilities of subsets of main memory (i.e., low physical addresses to be used by legacy DMA hardware) and we believe that they are more suitable to characterize the different memory tiers. To use NUMA nodes to describe memory tiers would mean that the locality information that gets conveyed with them, would be lost across the different memory types.

We reused the existing kswapd thread in Linux in charge of memory evictions for the implementation of the kernel thread that implements the demotion of pages. We also created a corresponding kernel thread, kpromoted, that is in charge of the promotion of pages by implementing similar logic to the one used for kswapd. These threads have an instance executing in the operating system for each available NUMA node. One question that arises is the potential benefit of using additional kernel threads to migrate pages between tiers. Our decision of using one thread per NUMA node comes from the granularity of existing locks protecting the node's data structures which maintain memory zones and page lists within. Careful restructuring of this locking granularity is necessary for future designs to benefit from using more than one threads per NUMA node.

Our approach relies on the access bit of memory pages for classification according to their frequency of accesses, as defined by the *importance* of the page. One possible improvement to this approach is to also include the dirtiness information for memory pages in a weighted formula to compute the *importance* of a page. By including this extra information we could weight the different types of accesses for a page (read or write) in the decision of page placement. This additional information becomes particularly relevant

when the underlying memory hardware exhibits non-uniform latency for the different types of accesses.

6.6 Summary

MULTI-CLOCK is a new approach to building systems for the next generation of data-intensive workloads by enabling the use of persistent memory as an extension of DRAM for use as main memory. MULTI-CLOCK seamlessly and dynamically tiers data across high-performance, high-cost, and high-power consuming DRAM with lower-performing, but also lower-cost and lower-power consuming persistent memory. Furthermore, because it does not rely on persistence capabilities, MULTI-CLOCK simplifies many of the reported overheads involved in using persistent memory. We evaluated a prototype implementation of MULTI-CLOCK by emulating persistent memory latency for a portion of DRAM using Intel's HMEP evaluation platform and running a variety of benchmark workloads. Our micro-benchmark evaluation reveals fairly low migration overheads of about 3.3% application performance degradation in the presence of continuous page migration across memory tiers. Our evaluation with PageRank, the graph analytics tool from GraphLab, yields results where MULTI-CLOCK, for a system with 64GB of DRAM and 192GB of persistent memory, reduces execution time by as much as 47% of a system with only persistent memory. MULTI-CLOCK's performance for this workload is also 37% better relative to a static tiering solution. Furthermore, MULTI-CLOCK achieves performance within 32% of an all-DRAM system, while static tiering suffers a significantly higher 88% degradation. With the YCSB workloads using a Memcached back-end, MULTI-CLOCK provides up to 10% higher throughput than the static tiering solution. Finally, while static tiering is unable to do so, MULTI-CLOCK implements dynamic fairness of resource allocation with two similarly configured instances of VoltDB. Our results demonstrates that MULTI-CLOCK is able to dynamically react to workload changes by utilizing the highest performance memory tier

for the hottest pages and thereby yielding significant performance improvement relative to a system that tiers data statically across memory tiers.

In the next chapter we discuss the body of related work to the solutions presented in this thesis.

6.7 Credits

Daniel Campello, Raju Rangaswami, and Andy Rudoff contributed to the preliminary design of *Managing Tiered Memory Systems with* MULTI-CLOCK. Daniel Campello executed all the experiments to obtain the results used to evaluate the use of byte-addressable persistent memory as an extension of DRAM in a multi-tier memory system with support for dynamic migration of pages according their access frequency.

CHAPTER 7

RELATED WORK

In this chapter we examine the body of related work to the systems that we presented in the previous chapters. The chapter is structured as follows. First we present work regarding clustering of virtual machine images. Next, we review previous work that, as non-blocking writes to files, aim to reduce or eliminate blocking in the context of the fetch-before-write problem. We conclude by looking into work that take advantage of persistent memory to improve system performance.

7.1 CORIOLIS: Scalable VM Clustering in Clouds

Clustering techniques and optimizations have been extensively studied in the past [RK87, LCGM+99, KMN+02, KMN+04]. These techniques rely on the efficiency of specific operations (e.g., distance, merge, etc.) over the elements to cluster to be computationally inexpensive and really fast. In our work, we address the problem of clustering VM images which inherently require very complex and computationally expensive operations to cluster.

Redundancy elimination based on identifying duplicate data is a popular topic of research [LEB+09, ZLP08, LJBR+16]. Finding similar clusters is a related problem but is more data intensive because it requires processing over the entire index of the data as well as a manifest linking images to their contents. Further, the data access for this problem does not have inherent data popularity and locality, which is used extensively by deduplication techniques for scaling.

The research work closest to ours is VMFlocks which applies standard deduplication techniques for images that are migrated together across data centers [AKSSR11]. Given a batch of images, It eliminates raw data duplicates across the given set of VM images.

However, it does not tackle identifying images with high redundancy or leveraging semantic similarity.

7.2 Non-blocking Writes to Files

Non-blocking writes have existed for almost three decades for managing CPU caches. Observing that entire cache lines do not need to be fetched on a word write-miss thereby stalling the processor, the use of additional registers that temporarily store these word updates was investigated [Kro81] and later adopted [LCBJ11].

Recently, non-blocking writes to main memory pages was motivated using full system memory access traces generated by an instrumented QEMU machine emulator [UKRV11]. This prior work outlined some of the challenges of implementing non-blocking writes in commodity operating systems. We improve upon this work by presenting a detailed design and Linux kernel implementation of non-blocking writes, addressing a host of challenges as well as uncovering new design points. We also present a comprehensive evaluation with a wider range of workloads and performance numbers from a running system.

A candidate approach to mitigate the fetch-before-write problem involves provisioning adequate DRAM to minimize write cache misses. However, the file system footprint of a workload over time is usually unpredictable and potentially unbounded. Alternatively, prefetching [SSS99] can reduce blocking by anticipating future memory accesses. However, prefetching is typically limited to sequential accesses. Moreover, incorrect decisions can render prefetching ineffective and pollute memory. Non-blocking writes is complementary to these approaches. It uses memory judiciously and only fetches those pages that are necessary for process execution.

There are several approaches proposed in the literature that reduce process blocking specifically for system call induced page fetches. The goal of the asynchronous I/O library (e.g., POSIX AIO [Ame94]) available on Linux and a few BSD variants is to make

file system writes asynchronous; a helper library thread blocks on behalf of the process. LAIO [ECCZ04] is a generalization of the basic AIO technique to make all system calls asynchronous; a library checkpoints execution state and relies on scheduler activations to get notified about the completion of blocking I/O operations initiated inside the kernel. Recently, FlexSC [SS10b] proposed asynchronous exception-less system calls wherein system calls are queued by the process in a page shared between user and kernel space; these calls are serviced asynchronously by syscall kernel threads which report completion back to the user process.

The scope of non-blocking writes in relation to the above proposals is different. Its goal is to entirely eliminate the blocking of memory writes to pages not available in the file system page cache. A non-blocking write does not need to checkpoint state thereby consuming lesser system resources. Further, it can be configured to be lightweight so that it does not use additional threads (often a limited resource in systems) to block on behalf of the running process. Finally, unlike these approaches which require application modifications to use specific libraries, non-blocking writes work seamlessly in the OS transparent to applications.

There are works that are related to non-blocking writes, but quite different in their accomplished goal. Speculative execution (or Speculator) as proposed by Nightingale *et al.* [NCF06] eliminates blocking when synchronously writing cached in-memory page modifications to a network file server using a process checkpoint and rollback mechanism. Xsyncfs [NVCF06] eliminates the blocking upon performing synchronous writes of in-memory pages to disk by creating a commit dependency for the write and allowing the process to make progress. Featherstitch [FMK+07] improves the performance of synchronous file system page updates by scheduling these page writes to disk more intelligently. Featherstitch employed patches but for a different purpose – to specify dependent changes across disk blocks at the byte granularity. OptFS [CPADAD13] decouples the or-

dering of writes of in-memory pages from their durability, thus improving performance. While these approaches optimize the writing of in-memory pages to disk they do not eliminate the blocking page fetch before in-memory modifications to a file page can be made.

BOSC [SLC12] describes a new disk update interface for applications to explicitly specify disk update requests and associate call back functions. Opportunistic Log [OS94] describes the fetch-before-write problem for objects and uses a second log to record updates. Both of these reduce application blocking allowing updates to happen in the background but they require application modification and do not support general-purpose usage. Non-blocking writes is complementary to the above body of work because it runs seamlessly inside the OS requiring no changes to applications.

7.3 Managing Tiered Memory Systems with MULTI-CLOCK

Emerging persistent memory technologies show promise in three distinct areas: non-volatility, very large capacity (as compared to DRAM cost), and performance suitable for direct load/store access by the CPU. Most studies on persistent memory focus on the non-volatility, using it to replace or extend block storage, implement persistent caches, or explore the persistent execution of processes that can survive power failures [CCA+11, CNF+09, DKK+14, KSDC14, YWC+15]. In contrast, our work focuses on the large capacity characteristic of persistent memory and the ability to read, write, and execute data residing in persistent memory. The study of Zhang et al. [ZS15] points out that the biggest performance overhead in the use of persistent memory is related to ensuring data resides safely in persistent memory, providing strong guarantees about non-volatility and consistency. Since our usage does not require non-volatility, the overhead required to maintain persistence and consistency across system interruption is avoided.

There have been many studies that explore the use of different types of memory for the building of *hybrid memory systems*. Such systems make use of the different characteristics

of the available memory types to combine them into a hybrid solution. Hybrid memory systems do not establish any specific hierarchy between the different memory types as tiered memory systems do. Qureshi et al. [QSR09] presents a hybrid memory system that uses both DRAM and PCM transparently to applications. In their work, DRAM is used as a buffer cache for data residing in PCM. In contrast we use both DRAM and persistent memory as separate tiers in a multi-tier memory system, where each tier holds a unique copy of the data that the CPU can access regardless of its location. Dhiman et al. [DAR09] and Ramos et al. [RGB11] both present hybrid memory systems where PCM is used as an extension of DRAM by modifying the memory controller in order to determine the placement of pages of memory. Our approach instead solves the problem by modifying the page replacement algorithm in the operating system without requiring any hardware modification.

Lee et al. [LKS+13] present a modified DRAM hardware architecture to obtain two types of DRAM with different latencies. Their approach to decide which pages to place in each type of memory relies on either static information using compiler-based profiling or dynamic information from hardware-based profiling. In contrast, our approach dynamically determines the best placement for memory pages without the need of code recompilation or hardware modifications.

Mogul et al. [MASF09] also present a hybrid memory system where flash is used as an extension of DRAM. Their approach, similar to ours, involves only operating system modifications to support this hybrid memory model in a transparent way to applications and without additional hardware support. In their work, flash is used as a read-only memory type, restricting any writes from applications. Writes are serviced by first migrating the page from flash to DRAM. In contrast, we allow both reads and writes to pages in any memory tier and present a promotion mechanism that can intelligently select the best can-

didates to be migrated to a higher tier depending on their access frequency. Additionally, our approach can be easily extended to cover more than two types of main memory devices.

Bock et al. [BCMM14], assume an existing hybrid memory system in mobile devices and tackle the orthogonal problem of page migration efficiency. Their work is complimentary to ours and can be used together with our solution to improve the performance of applications affected by page migrations.

Many replacement algorithms have been studied in the past in the context of caching [KA08, MM04, PJK+06, WW02, ZPL01]. Our solution is orthogonal to these efforts and builds upon existing memory replacement mechanisms and presents a modified page migration and replacement algorithm for tiered memory.

CHAPTER 8

CONCLUSIONS

Modern computing systems use expensive CPU-addressable memory to maintain application's temporary data and to cache copies of persistent data stored in slow secondary storage. In this thesis we presented three approaches to improve overall system performance by optimizing main memory usage.

First, we tackled the issue of VM placement in virtualized systems where different VMs share physical resources (including main memory) on the host machines. We described the CORIOLIS framework and system that was specifically designed for scalable clustering of VM images to intelligently place VM into hosts and, with the aid of deduplication techniques, to optimize memory usage and overall system performance. Next, we focused on how to improve the complex machinery employed by caches for managing their cached data. Writing data to a page not present in the file system page cache causes the operating system to synchronously fetch the page into memory, always blocking the writing process. Our *non-blocking writes* approach to handling writes eliminates such blocking by buffering the written data elsewhere in memory and unblocking the writing process immediately. This buffering allowed the system to service file writes in a faster way and, in some configurations, with less memory resources when accesses incur in a *cache miss*. Last, we investigated the use of emerging byte-addressable persistent memory technology as a less costly alternative to expensive DRAM for extending system's main memory. We motivated and built a tiered memory system that achieves improved application performance at lower cost and power consumption with the goal of placing the right data in the right memory tier and at the right time.

As we move into the future more and more workloads will move to main memory displacing secondary storage to a less important role in the system. How well a system makes use of the available CPU-addressable memory will most likely dictate the performance of

the whole system. In conclusion, we believe that the work presented in this thesis has the impact of improving today's systems performance by using main memory in a more effective way. The elimination of process blocking of file writes, the deduplication of intelligently clustered VM's on a host, and the inclusion of extra main memory capacity through the use of persistent memory are just the starting point, as we hope to set a path for future research in this area.

BIBLIOGRAPHY

[ABM+11] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang. Virtual machine images as structured data: the Mirage image library. In *Proc. HotCloud*, 2011.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*, OLS '09, July 2009.

[AKSSR11] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. VMFlock: Virtual Machine Co-migration for the Cloud. In *Proc. of the IEEE/ACM HPDC*, June 2011.

[Ame94] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. IEEE, 1994. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[AMR09] D. Arthur, B. Manthey, and H. Roeglin. k-means has polynomial smoothed complexity. In *IEEE FOCS*, 2009.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall Press, 1st edition, 1986.

[BCMM14] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mosse. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '14, May 2014.

[BLM+12] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage*, MSST '12, April 2012.

[CAVL09] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proc. of the USENIX ATC*, June 2009.

[CCA+11] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[CCV+13] Daniel Campello, Carlos Crespo, Akshat Verma, Raju Rangaswami, and Praveen Jayachandran. Coriolis: Scalable VM clustering in clouds. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '13, June 2013.

[chr] Reduce Chrome memory. https://support.google.com/chrome/answer/6152583.

[CKZ11] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, May-June 2011.

[CLU+15] Daniel Campello, Hector Lopez, Luis Useche, Ricardo Koller, and Raju Rangaswami. Non-blocking writes to files. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '15, February 2015.

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '09, 2009.

[CPADAD13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, November 2013.

[Cro13] Crossbar Resistive Memory. The Future Technology for NAND Flash. http://www.crossbar-inc.com/assets/img/media/Crossbar-RRAM-Technology-Whitepaper-080413.pdf, 2013.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the ACM symposium on Cloud computing*, SoCC '10, June 2010.

[DAR09] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: a hybrid pram and dram main memory system. In *Proceedings of the Annual Design Automation Conference*, 2009.

[DAX] Direct Access for Files. http://www.kernel.org/doc/Documentation/filesystems/dax.txt.

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

[DKK+14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems*, EuroSys '14, 2014.

[DN65] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 213–229, 1965.

[DRZ+16] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the European Conference on Computer Systems*, EuroSys '16, April 2016.

[DSL10] B. Debnath, S. Sengupta, and J. Li. ChunkStash: speeding up inline storage deduplication using flash memory. In *Usenix ATC*, 2010.

[ECCZ04] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the 2004 USENIX Annual Technical Conference*, ATC '04, 2004.

[ELMS03] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '03, 2003.

[EMC12] EMC. VFCache. http://www.emc.com/storage/vfcache/vfcache.htm, 2012.

[ESO13] ESOS Laboratory. Mobibench traces. https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen, 2013.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. In *Linux Journal*, 2004.

[FMK+07] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, pages 307–320, October 2007.

[Fus12] Fusion-IO. ioTurbine. http://www.fusionio.com/systems/ioturbine/, 2012.

[GMC+12] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference*, ATC '12, June 2012.

[GP11] Al Guillen and Randy Perry. Understanding Linux Deployment Strategies: The Business Case for Standardizing on Red Hat Enterprise Linux. http://www.redhat.com/f/pdf/StandardizeOnRHEL3.pdf, April 2011.

[GPG+11] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent-based dynamic tiering. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '11, February 2011.

[Hag87] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '87, November 1987.

[HDV+11] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '11, October 2011.

[JLH+13] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. Framework for analyzing android I/O stack behavior: From generating the workload to analyzing the trace. *Future Internet*, 5(4):591–610, 2013.

[JM09] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of SysStor*, 2009.

[JPZ+11] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. An empirical analysis of similarity in virtual machine images. In *ACM Middleware*, 2011.

[KA08] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *USENIX File and Storage Systems (FAST)*, 2008.

[KAU12] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '12, February 2012.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the International World Wide Web Conference*, WWW '10, April 2010.

[KM06] Taeho Kgil and Trevor Mudge. FlashCache: a NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, October 2006.

[KMN+02] T. Kanungo, D. M. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. In *IEEE Trans. Pattern Analysis and Machine Intelligence*, pages 881–892, 2002.

[KMN+04] T. Kanungo, D. M. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. Local search approximation algorithm for k-means clustering. In *Computational Geometry: Theory and Applications*, pages 89–112, 2004.

[KMR+13] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '13, February 2013.

[KMR15] Ricardo Koller, Ali Jose Mashtizadeh, and Raju Rangaswami. Centaur: Hostside SSD caching for storage performance control. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '15, July 2015.

[KR10] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/ O Performance. In *Proc. of USENIX FAST*, 2010.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 81–87. IEEE Computer Society Press, 1981.

[KSDC14] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, February 2014.

[KVR10] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation*, 67(11):1139–1154, 2010.

[LCBJ11] Sheng Li, Ke Chen, Jay B. Brockman, and Norman P. Jouppi. Performance impacts of non-blocking caches in out-of-order processors. Technical report, Hewlett-Packard Labs and University of Notre Dame, July 2011.

[LCGM+99] Chen Li, Edward Chang, Hector Garcia-Molina, James Wang, and Gio Wilderhold. Clindex: Clustering for similarity queries in high-dimensional spaces. *Stanford Technical Report SIDL-WP-1998-0100*, Feb. 1999.

[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 *USENIX FAST*, 2009.

[LJBR+16] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, February 2016. USENIX Association.

[LKS+13] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, 2013.

[Lov10] Robert Love. *Linux Kernel Development*. Pearson Education, Inc., 2010.

[LPGM08] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file systemworkloads. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[MASF09] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS '09, 2009.

[MB11] Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *Proc. of USENIX FAST*, February 2011.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*, pages 163, 196. Addison Wesley, 1996.

[MDS+15] J. Malicevic, S. R. Dulloor, N. Sundaram, N. Satish, J.Jackson, and W. Zwaenepoel. Exploiting NVM in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, October 2015.

[MFG+12] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. KSM++: Using I/O based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, RESoLVE '12, March 2012.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970.

[Mit13] Sparsh Mittal. Energy saving techniques for phase change memory (PCM). In *arXiv:1309.3785*, 2013.

[MM04] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[MSS12] Alexander Makarov, Viktor Sverdlov, and Siegfried Selberherr. Modeling emerging non-volatile memories: Current trends and challenges. In *Proceedings of the International Conference on Solid State Devices and Materials Science*, SSDM '12, April 2012.

[NCF06] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems*, pages 361–392, 2006.

[Net13] NetApp. Flash Accel. http://www.netapp.com/us/products/storage-systems/flash-accel/, 2013.

[NVCF06] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation*, OSDI '06, November 2006.

[ODCH+85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD/ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '85, 1985.

[OS94] J. O'Toole and L. Shrira. Opportunistic log: Efficient installation reads in a reliable storage server. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, OSDI '94, pages 39–48, 1994.

[PADAD05] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference*, ATC '05, June 2005.

[PJK+06] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, October 2006.

[PVO+15] Mahesh Patil, Murali Vilayannur, Michal Ostrowski, Sameer Narkhede, Venkatesh Kothakota, Woon Jung, Heiko Kohler, Govindarajan Soundararajan, Kaustubh Patil, Chethan Kumar, and Deepavali Bhagwat. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, 2015.

[QSR09] Moinuddin K. Qureshi, Viji Srinivasan, and Jude A. Rivers. Scalable high-performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '09, 2009.

[RA00] Drew Roselli and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, ATC '00, 2000.

[RGB11] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the 25th International Conference on Supercomputing*, ICS '11, 2011.

[RK87] LKPJ Rousseeuw and L Kaufman. Clustering by means of medoids. *Statistical data analysis based on the L1-norm and related methods*, 405, 1987.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17:365–375, July 1974.

[Sal] Salvatore Sanfilippo and Pieter Noordhuis. Redis. http://redis.io.

[sap] SAP HANA. http://hana.sap.com.

[SK12] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, June 2012.

[SLC12] Dilip Nijagal Simha, Maohua Lu, and Tzi-cker Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[spa] Apache Spark. http://spark.apache.org.

[SS10a] Mohit Saxena and Michael M. Swift. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of the USENIX Annual Technical Conference*, ATC '10, June 2010.

[SS10b] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8. USENIX Association, 2010.

[SSS99] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why does file system prefetching work? In *Proceedings of the USENIX Annual Technical Conference*, ATC '99, 1999.

[Sta08] Standard Performance Evaluation Corporation. SPECsfs2008. http://www.spec.org/sfs2008/, 2008.

[SUN11] SUN Microsystems. Filebench 1.4.9.1. http://sourceforge.net/projects/filebench/, 2011.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[Tra] Transaction Processing Performance Council (TPC). TPC-C Benchmark. http://www.tpc.org/tpcc.

[UKRV11] Luis Useche, Ricardo Koller, Raju Rangaswami, and Akshat Verma. Truly non-blocking writes. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '11, June 2011.

[VMW12] VMWare. VMWare vSphere Data Protection. Technical White Paper, June 2012.

[VMw13] VMware, Inc. VMware Virtual SAN. http://www.vmware.com/products/virtual-san/, 2013.

[Vol] VoltDB. VoltDB 5.0.1 Community Edition. http://voltdb.com.

[VVK+12] B. Viswanathan, A. Verma, B. Krishnamurthy, P. Jayachandran, K. Bhattacharya, and R. Ananthanarayanan. Rapid adjustment and adoption to MIaaS clouds. In *ACM Middleware, Industry track*, 2012.

[Wal02] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proc. Usenix OSDI*, 2002.

[WR10] Xiaojian Wu and A. L. Narasimha Reddy. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *Proceedings of the IEEE International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, September 2010.

[WW02] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '02, June 2002.

[ycs] YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[YWC+15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '15, February 2015.

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *USENIX FAST*, 2008.

[ZPL01] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, USENIX ATC '01, June 2001.

[ZS15] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies*, MSST '15, June 2015.

VITA

DANIEL CAMPELLO

	Born, Los Teques, Venezuela
2009	B.Eng., Computer Engineering Universidad Simon Bolivar Sartenejas, Venezuela
2012	M.S., Computer Science Florida International University Miami, Florida
2010–2015	Graduate Research Assistant Florida International University Miami, Florida

PUBLICATIONS

Guerra, J., Marmol, L., Campello, D., Crespo, C., Rangaswami, R., (2012). *Software Persistent Memory*. USENIX Annual Technical Conference, 319–331.

Campello, D., Crespo, C., Verma, A., Rangaswami, R., Jayachandran, P., (2013). *Coriolis: Scalable VM Clustering in Clouds*. ICAC, 101–105.

Campello, D., Lopez, H., Useche, L., Koller, R., Rangaswami, R., (2015). *Non-blocking Writes to Files*. USENIX Conference on File and Storage Technologies, 151–165.