FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

OPTIMIZING STORAGE AND MEMORY SYSTEMS FOR ENERGY AND PERFORMANCE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Luis Useche

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Luis Useche, and entitled Optimizing Storage and Memory Systems for Energy and 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

Ming Zhao

Kaushik Dutta

Ajay Gulati

Raju Rangaswami, Major Professor

Date of Defense: July 16, 2012

The dissertation of Luis Useche is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

DEDICATION

To Barbi and Elsi

ACKNOWLEDGMENTS

This dissertation would not have been possible without the collaboration, support, and help of many people. I want to thank every person that in one way or another helped me in the last six years to complete my work.

I cannot thank enough my wife, Barbara, for all her love, care, encouragement, and even dissertation related discussions throughout this period. Without you, this big step in my life would have been unachievable.

I want to thank my advisor, Prof. Raju Rangaswami, for giving me this chance and guiding me through this journey. Raju, thanks for always motivating me when I needed it the most. I very much enjoyed the brainstorming sessions and mini-pcs we always had. I am very grateful for all your mentoring and specially for having the patience to teach a non-English speaker how to write and speak in English. For this and much more, thank you.

A big thanks goes to my co-authors: Ricardo Koller, Jorge Guerra, Akshat Verma, and Jesus Ramos for helping me with my projects at FIU. Especially, I want to thank Ricardo Koller for helping me with many intellectual discussions, code sessions, and experiments.

I want to thank all my dissertation committee for their insightful feedback that made this work much better.

I wish to thank all the people in the department, especially Olga, for helping me with all the bureaucratic paperwork that a Ph.D. requires.

To all of you, thanks.

ABSTRACT OF THE DISSERTATION

OPTIMIZING STORAGE AND MEMORY SYSTEMS FOR ENERGY AND PERFORMANCE

by

Luis Useche

Florida International University, 2012

Miami, Florida

Professor Raju Rangaswami, Major Professor

Electrical energy is an essential resource for the modern world. Unfortunately, its price has almost doubled in the last decade. Furthermore, energy production is also currently one of the primary sources of pollution. These concerns are becoming more important in data-centers. As more computational power is required to serve hundreds of millions of users, bigger data-centers are becoming necessary. This results in higher electrical energy consumption. Of all the energy used in data-centers, including power distribution units, lights, and cooling, computer hardware consumes as much as 80%. Consequently, there is opportunity to make data-centers more energy efficient by designing systems with lower energy footprint. Consuming less energy is critical not only in data-centers. It is also important in mobile devices where battery-based energy is a scarce resource. Reducing the energy consumption of these devices will allow them to last longer and re-charge less frequently.

Saving energy in computer systems is a challenging problem. Improving a system's energy efficiency usually comes at the cost of compromises in other areas such as performance or reliability. In the case of secondary storage, for example, spinning-down the disks to save energy can incur high latencies if they are accessed while in this state. The challenge is to be able to increase the energy efficiency while keeping the system as reliable and responsive as before.

This thesis tackles the problem of improving energy efficiency in existing systems while reducing the impact on performance. First, we propose a new technique to achieve fine grained energy proportionality in multi-disk systems; Second, we design and implement an energy-efficient cache system using flash memory that increases disk idleness to save energy; Finally, we identify and explore solutions for the page *fetch-before-update* problem in caching systems that can: (a) control better I/O traffic to secondary storage and (b) provide critical performance improvement for energy efficient systems.

TABLE OF CONTENTS

CHAPTER PAGE

1. Introduction 1

2. Problem Statement 5
2.1 Thesis Statement 5
2.2 Thesis Statement Description 5
2.3 Thesis Significance 9

3. Energy Proportional Storage 11
3.1 Proportionality Matters 11
3.2 Design Goals 13
3.3 Storage Workload Characteristics 15
3.4 Background and Rationale 19
3.5 Design Overview 22
3.5.1 Load Monitor 23
3.5.2 Replica Placement Controller 24
3.5.3 Active Disk Manager 24
3.5.4 Consistency Manager 25
3.5.5 Replica Manager 25
3.6 Algorithms and Optimizations 26
3.6.1 Replica Placement Algorithm 26
3.6.2 Active Disk Identification 30
3.6.3 Key Optimizations to Basic SRCMap 32
3.7 Evaluation 34
3.7.1 Prototype Results 37
3.7.2 Simulator Results 40
3.7.3 Resource overhead of SRCMap 45
3.8 Summary 46
3.9 Credits 46

4. Energy-efficient Storage using Flash 47
4.1 Overview 47
4.2 Profiling Energy Consumption 50
4.3 System Architecture 51
4.4 System Design 53
4.4.1 Page Access Tracker 53
4.4.2 Indirection 54
4.4.3 Reconfiguration Trigger 56
4.4.4 Reconfiguration Planner 57
4.4.5 Reconfigurator 58
4.4.6 Other Design Issues 59

4.5 System Implementation . . . 60
4.5.1 Maintaining the Top-k Ranked Pages . . . 60
4.5.2 Indirector Implementation Issues . . . 62
4.5.3 Modularization and Consistency . . . 63
4.6 Evaluation . . . 64
4.6.1 Choosing the Disk Spin-down Timeout . . . 66
4.6.2 Energy Savings . . . 67
4.6.3 Performance Impact of External Caching . . . 69
4.6.4 Resource Overhead . . . 70
4.7 Summary . . . 72
4.8 Credits . . . 73

5. Controlling I/O Traffic with Non-blocking Writes . . . 74
5.1 The Fetch-before-update Behavior . . . 74
5.2 Motivating Non-blocking Writes . . . 77
5.2.1 Solution Impact . . . 78
5.3 Non-blocking Writes . . . 82
5.3.1 Approach Overview . . . 83
5.3.2 Write Interposition . . . 85
5.3.3 Page Patching . . . 88
5.3.4 Non-blocking Reads . . . 89
5.3.5 Scheduling with Non-blocking writes . . . 90
5.4 Optimizations . . . 93
5.4.1 Alternative Page Fetching Modes . . . 93
5.4.2 To Fetch or Not to Fetch and When . . . 96
5.5 Correctness . . . 97
5.6 Estimating Benefits . . . 100
5.6.1 Virtual Memory Simulation . . . 101
5.6.2 Fraction of Non-blocking Write Faults . . . 102
5.6.3 Outstanding Write Fetches . . . 102
5.6.4 Estimating Overall Savings . . . 103
5.7 Evaluation . . . 104
5.7.1 Experimental setup . . . 104
5.7.2 Performance Improvements . . . 106
5.7.3 Memory Sensitivity . . . 110
5.7.4 Optimizations with Patches . . . 111
5.8 Summary . . . 112
5.9 Credits . . . 113

6. Related Work . . . 114
6.1 Energy Proportionality in Storage Systems . . . 114
6.2 Energy Efficient Storage with Flash . . . 117
6.3 *fetch-before-update* Problem . . . 119

7. Conclusions 122

8. Future Work 125

BIBLIOGRAPHY 130

VITA 139

LIST OF TABLES

TABLE PAGE

3.1 Summary statistics of one week I/O workload 15

3.2 Workload and storage system details. 35

3.3 SRCMap experimental settings 37

4.1 Various laptop configurations used in profiling experiments. 50

4.2 Specifications of the machines used in the experiments 66

4.3 EXCES memory overhead 71

5.1 Time benefit estimation for non-blocking writes 79

5.2 Full system memory traces workloads 100

5.3 Non-blocking writes evaluation workloads 105

5.4 Performance improvements due to patch optimizations 111

6.1 Comparison of Power Management Techniques 114

LIST OF FIGURES

FIGURE PAGE

2.1 Diagram depicting the thesis contributions 9

3.1 Variability in I/O workload intensity 16

3.2 Overlap in daily working sets 17

3.3 Distribution of read-idle times 18

3.4 SRCMap integrated into a Storage Virtualization Manager 22

3.5 Replica Placement Model 28

3.6 Active Disk Identification 29

3.7 Logical view of experimental setup 35

3.8 Power and active disks time-line. 38

3.9 Impact of consolidation on response time. 39

3.10 SRCMap prototype results 41

3.11 Load and power consumption for each disk. 42

3.12 Sensitivity to over-provisioned space. 44

3.13 Energy proportionality with load. 45

4.1 Energy consumption profiles of various ECD types and interfaces. 50

4.2 EXCES system architecture. 52

4.3 Page rank decay function 54

4.4 Indirection example 56

4.5 The `page_ranker` structure 60

4.6 Example Top-k matrix 61

4.7 Alignment problem example 62

4.8 Effect of the disk spin-down timeout value on energy savings 66

4.9 Energy consumption with different workloads 67

4.10 Performance impact of EXCES with various workloads 70

5.1 A non-blocking write in action 75
5.2 Page fetch asynchrony with non-blocking writes. 78
5.3 Estimate of fetches that benefit from non-blocking writes 79
5.4 Page fetch parallelism with non-blocking writes 82
5.5 State diagram for out-of-core page access. 83
5.6 State diagram for out-of-code page access with non-blocking writes. . . . 84
5.7 Example of non-blocking writes scheduling problem 90
5.8 Current and new process state diagrams 92
5.9 A non-blocking write with lazy fetch. 94
5.10 A non-blocking write with scheduled fetch. 94
5.11 Expected OWF for various workloads 102
5.12 Execution time change in single-threaded applications 106
5.13 Execution time change in multi-threaded applications 107
5.14 SPEC SFS2008 and SPEC Power2008 response times 109
5.15 Sensitivity of non-blocking writes performance with memory size 110

CHAPTER 1

INTRODUCTION

The U.S. Enviromental Protection Agency reported in 2007 that the energy cost of U.S. Government servers and data-centers was $450 million in 2006 [Age07]. This cost may be higher in the future if electrical energy prices continue to increase at the same rate as the last decade: 45% in total [Adm11]. Barroso et al. estimate that 50% of the energy used by data-centers is consumed by IT hardware [BH09]. In highly efficient data-centers, this number has increased to 80% [DMR+11]. Reducing the energy consumption of computer systems will have an important impact on data-center cost. Saving energy is important not only in data-centers. It is also important in mobile devices where battery-based energy is a scarce resource. Reducing the energy footprint on these devices will allow them to last longer, re-charge less frequently, and replace batteries less often.

In this dissertation we address two of the most energy consuming devices in computer systems: memory and disk. In data-centers, memory and disk account for 30% and 10% of the total hardware energy consumption respectively [BH09]. On mobile devices, memory and disk usually are the most energy consuming devices after display and CPU with up to 6% and 15% share of the overall energy consumption [MV04].

Unfortunately, reducing the energy consumption on data-centers and mobile devices is not easy task. Saving energy usually trades-off performance for energy efficiency. Hence, designing energy-efficient systems must go beyond reducing their energy footprint and take into account performance implications. DRAM, for instance, continuously consumes energy in order to refresh its banks of memory even when not in use [DMR+11] (static energy consumption). DRAM's static energy consumption can be reduced if a portion of memory is replaced with a slower but

bigger and more energy efficient flash. Although energy will be saved, this would induce higher paging activity and, consequently, increase the latency of accesses to data. As a second example let us consider hard disk drives. They can only save energy while spun-down. While applications usually do not offer many periods of idleness, requests to disks that are spun-down can incur delays of several seconds (up to 10) while the disk starts spinning up again. These two examples illustrate the energy-performance trade-off challenge that should be considered when designing energy efficient systems. Moreover, it also highlights the importance of complementary performance optimizations as a critical catalyst to increase the viability of energy efficient systems in production deployments.

Researchers have tackled these challenges using various methods. On the storage side, previous work has provided techniques to create various levels of energy consumption in multi-disks systems despite their individual two energy level limitations [WOQ+07]. Moreover, researchers have also explored keeping disks inactive for long periods of time by adding energy-efficient flash to cache popular data [MDK94, CJZ06] or off-loading writes to a single disk in multi-disks systems [NDR08]. For memory, researchers have explored keeping a large portion of DRAM inactive by increasing locality in accesses [SCN+10]. Others have proposed to add more power-levels to DRAM given that applications usually do not use their peak bandwidth [DMR+11].

In this thesis, we address these challenges by exploring three complementary directions. First, we design and evaluate a new low-overhead fine-grained energy proportional multi-disk storage system. Disks are unable to consume power proportionally to their load due to their small number of power modes. We propose and evaluate a new energy proportional multi-disk storage system called SRCMap (see §3). SRCMap selectively replicates blocks among disks to create multiple sources for

the same data. Then, as less load is received, SRCMap spins-down disks whose data can be served from another device. On the other hand, when the load increases, SRCMap spins-up disks as needed.

Second, we design, implement, and evaluate an external cache system to increase disk idleness and save energy. Applications typically do not contain long idle periods of time that allow disks to spin-down and save energy. We design and implement a new external caching storage system that caches popular data to create long idle periods of time and lets the disk spin-down (see §4). This energy-efficient external device absorbs requests that would otherwise need to be serviced from disk, interrupting its low-power state mode.

Finally, we identify the *fetch-before-update* problem in caching systems and present a solution and implementation to evaluate its benefits. *Fetch-before-update* refers to the behavior of commodity operating systems where a page fetch from disk is required when an application updates an out-of-core page. Such fetches block applications while the I/O is completed resulting in an execution slowdown. This occurs mainly due to the differences in access granularity of memory and disk. *Fetch-before-update* can be eliminated by temporarily buffering the update elsewhere in memory while the I/O is being performed, and merging it in once the page is in memory. We explore this new solution to the *fetch-before-update* problem as an opportunity to better control I/Os to secondary storage (see §5). Filtering I/Os to inactive disks will help to keep them in low-power state longer. Moreover, this solution has the potential to maintain performance when system memory is reduced for energy efficiency purposes.

This thesis is organized as follows. Chapter 2 states the dissertation problem and gives an overview of each of the solutions being proposed. Chapters 3, 4, 5 detail

each of the solutions. Chapter 6 presents related work. Chapter 7 draws conclusions from this work. Finally, Chapter 8 presents future research directions.

CHAPTER 2

PROBLEM STATEMENT

In this chapter we detail each of the problems and solutions we address in this dissertation. First, we provide a formal thesis statement. Next, we describe in more detail each of the problems and overview our proposed solutions. Finally, we describe the impact that this dissertation has in the energy-efficient systems area and how this benefits current systems.

2.1 Thesis Statement

In this dissertation we aim to reduce the energy consumption of computer systems by pursuing three complementary research directions:

- designing a multi-disk energy proportional storage system using commodity devices,
- designing and implementing energy-efficient storage systems using flash devices as an external cache to reduce energy consumption in disks, and
- designing and implementing a new mechanism to eliminate blocking read page operations due to page writes that could, otherwise, disturb disks in their low-power state and slowdown applications.

2.2 Thesis Statement Description

This thesis starts by addressing the energy consumption of disks disproportional to their load. Commodity disks can only handle requests while operating in their active state even if the accesses require only a small fraction of the maximum bandwidth provided by the device. This problem is especially important in multi-disk systems where the aggregate bandwidth provisioned is typically much higher than

the average load to mitigate peaks in load[BH09]. Hence, creating a solution to use energy proportional the load has the potential to save energy in multi-disks systems provisioned for peak load.

Multi-disks systems offer the possibility of serving data from a subset of disks when their maximum performance is not required. An ideal solution should have low space overhead and support heterogeneous devices with different performance-power ratio.

In our first contribution, we investigate the problem of creating an energy proportional storage system through power-aware dynamic storage consolidation. Our proposal, Sample-Replicate-Consolidate Mapping (SRCMap), is a storage virtualization layer optimization that enables energy proportionality for dynamic I/O workloads by consolidating the cumulative workload on a subset of physical volumes proportional to the I/O workload intensity. Instead of migrating data across physical volumes dynamically or replicating entire volumes, both of which are prohibitively expensive, SRCMap samples a subset of blocks from each data volume that constitutes its working set and replicates these on other physical volumes. During a given consolidation interval, SRCMap activates a minimal set of physical volumes to serve the workload and spins down the remaining volumes, redirecting their workload to replicas on active volumes. We present both theoretical and experimental evidence to establish the effectiveness of SRCMap in minimizing the energy consumption of enterprise storage systems. We present all the details of SRCMap in Section 3 including the design and evaluation of the system based on emulation.

Single-disks systems include embedded systems and portable computers. Saving energy on these devices allow them to last longer with the same energy charge. Unlike multi-disk systems, single-disk systems, having only two power modes, do not provide the benefit of proportionally adapting to the access load. Moreover,

the disk has to be completely undisturbed to save energy. Consequently, operating systems have to do their best and create long periods of idle time for disks. Usually, this is done by keeping popular data in higher level caches to absorb all the load that disks will observe otherwise. Extending this cache with a persistent low-energy *external cache device* (ECD), like flash, will allow for longer idle time as more cache is available to disks that can absorb most of the I/O operations. In our second contribution, we designed and implemented EXCES, an external caching system that employs prefetching, caching, and buffering of disk data for reducing disk activity. EXCES addresses important questions related to external caching, including the estimation of future data popularity, I/O indirection, continuous reconfiguration of the ECD contents, and data consistency.

We evaluated EXCES with both micro- and macro-benchmarks that address idle, I/O intensive, and real-world workloads. Overall system energy savings were found to lie in the modest 2-14% range, depending on the workload, in somewhat of a contrast to the higher values predicted by earlier studies. Furthermore, while the CPU and memory overheads of EXCES were well within acceptable limits, we found that flash-based external caching can substantially degrade I/O performance. We believe that external caching systems hold promise. Further improvements in flash technology, both in terms of their energy consumption and performance characteristics can help realize the full potential of such systems. Section 4 details our design, implementation, and results for EXCES.

Both of the systems we proposed above may have performance degradation when applications access data not available in the active devices: disks that are active in SRCMap, or flash in EXCES. While this accesses are unavoidable for reads, we found that it is possible to delay or even eliminate them for writes. Writing data to an out-of-core page causes the operating system to first fetch the page into memory before

it can be written into. Such *fetch-before-update* behavior incurs in unnecessary I/O operations that can wake up a disk in low-power state, thus, increasing energy usage. Moreover, *fetch-before-update* can also degrade performance by blocking the writing process during the page fetch operation. *Fetch-before-update* can especially hurt performance on energy efficient systems with low memory and high paging activity. In our final contribution, we develop non-blocking writes which delays such I/Os by buffering the written data temporarily elsewhere in memory and unblocking the writing process immediately. Non-blocking writes has the opportunity to delay the fetch operation to a later time (e.g. when the disk is active again), or issue it asynchronously to be handled immediately. Once the I/O request is done, the updates are applied to the page. The benefits of non-blocking writes are two-fold. It can save energy by delaying read accesses due to *fetch-before-update* on sleeping disks. Or it can allow processes to overlap computation with I/O to a greater extent and improve the parallelism of page fetch operations leading to greater page fetch throughput from storage. Notably, non-blocking writes works seamlessly inside the OS requiring no changes to applications. When used for proportionality, non-blocking writes will avoid spinning up a disk when the data is not available in the active disks. In case of flash caching for disks, when the block written is not in flash we can use non-blocking writes to hold the data until the disk is active again. We present our design, implementation, and evaluation of non-blocking writes in Chapter 5.

We summarize all these contributions in Figure 2.1. The figure shows the areas we improve: energy in multi-disk systems with SRCMap, energy in single-disk systems with EXCES, and performance with non-blocking writes. Additionally, it depicts possible improvements that we discuss in the chapters to follow. For example, we explain how non-blocking writes can help to increase the effectiveness

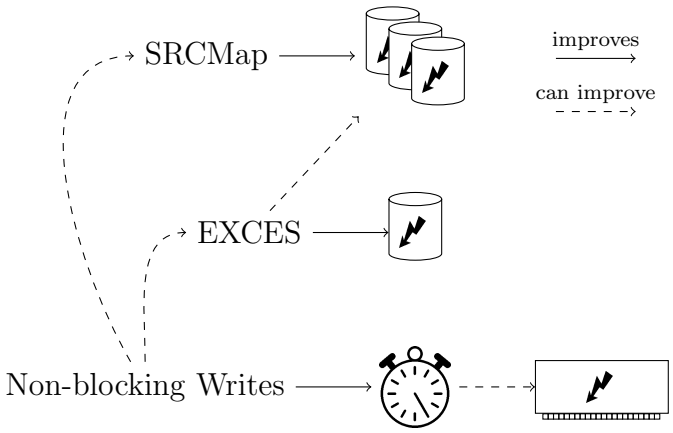


Figure 2.1: Diagram depicting the thesis contributions. It shows the areas we improve as well as the areas we have the potential to enhance. The lightning ( ) represents the energy of various systems: multi-disk, single-disk, and memory systems. The watch ( ) represents performance.

of energy efficient caching systems like SRCMap and EXCES as well as how the performance improvement of non-blocking writes can be used to reduce the energy used by DRAM.

2.3 Thesis Significance

The ideas proposed in this dissertation can be used seamlessly in data-centers and personal computing to save energy. They also improve the viability of energy efficient systems by closing their performance gap with commodity systems. Since all our systems are implemented at the operating system level or below, applications do not have to be modified. This will allow users to immediately avail of more energy efficient systems without changing the way they use computers.

With our new energy proportional storage system, data-centers do not have to worry about using more energy when over-provisioning to prepare for high peak loads. With only a few configuration knobs, SRCMap will automatically use the

available resources, increasing the energy consumption only when the load is higher. In data-centers, this has the advantage of freeing the administrator from energy concerns while at the same time lowering the energy costs.

For single-disk systems like portable computers, EXCES allows users to increase energy efficiency of their mobile devices by simply plugging in an energy efficient external device. This could result in new mobile devices that already include an internal flash to make batteries last longer. This contribution can also help in reducing the weight of mobile devices by reducing the battery size.

Solving the *fetch-before-update* behavior helps to increase the feasibility of SR-CMap and EXCES by keeping disks longer in their energy saving state and reduced their performance impact. Consequently, it will increase the energy efficiency without adding new hardware. This will allow executing more tasks with the same energy consumption as before. Finally, it will also reduce the response times of accesses by decreasing their dependency on the performance of the backing store.

By using caching techniques we are able to save energy in both multi-disk systems—mostly available in data-centers—and single-disks systems–available in portable systems. We also present solutions to control I/O activity that could save energy in caching systems, increase the performance in low-memory but energy efficient systems, and increase performance in general. Overall, the solutions proposed in this dissertation substantially advance the state of the art in the energy efficiency of data-centers and portable devices.

CHAPTER 3

ENERGY PROPORTIONAL STORAGE

We have seen in Chapter 2 that disks consume the same amount of energy regardless of their current load. This is particularly troublesome for data centers, where plenty of disks are typically provisioned in order to handle peak loads, whereas a fraction of them is needed to meet service levels in the average case. In this chapter we present the design and evaluation of a solution for achieving energy proportionality in multi-disk systems with a low overhead. This solution has the potential to improve the energy efficiency in data centers for the average load without hurting the system performance during peak loads.

3.1 Proportionality Matters

Energy Management has emerged as one of the most significant challenges faced by data center operators. The current power density of data centers is estimated to be in the range of 100 W/sq.ft. and growing at the rate of 15-20% per year [HP06]. Barroso and Hölzle have made the case for energy proportional computing based on the observation that servers in data centers today operate at well below peak load levels on an average [BH07]. A popular technique for delivering energy proportional behavior in servers is consolidation using virtualization [BKB07, TWM+08, VAN08, VDN+09]. These techniques (a) utilize heterogeneity to select the most power-efficient servers at any given time, (b) utilize low-overhead live Virtual Machine (VM) migration to vary the number of active servers in response to workload variation, and (c) provide fine-grained control over energy consumption by allowing the number of active servers to be increased or decreased one at a time.

Storage consumes roughly 10-25% of the power within computing equipment at data centers depending on the load level, consuming a greater fraction of the energy

when server load is lower [BH09]. Energy proportionality for the storage subsystem thus represents a critical gap in the energy efficiency of future data centers. In this work, we the investigate the following fundamental question: *Can we use a storage virtualization layer to design a practical energy proportional storage system?*

Storage virtualization solutions (e.g., EMC Invista [EMC], HP SVSP [Cor], IBM SVC [IBM], NetApp V-Series [Net]) provide a unified view of disparate storage controllers thus simplifying management [IDC06]. Similar to server virtualization, storage virtualization provides a transparent I/O redirection layer that can be used to consolidate fragmented storage resource utilization. Similar to server workloads, storage workloads exhibit significant variation in workload intensity, motivating dynamic consolidation [LPGM08]. However, unlike the relatively inexpensive VM migration, migrating a logical volume from one device to another can be prohibitively expensive, a key factor disrupting storage consolidation solutions.

Our proposal, Sample-Replicate-Consolidate Mapping (SRCMap), is a storage virtualization layer optimization that makes storage systems energy proportional. The SRCMap architecture leverages storage virtualization to redirect the I/O workload without any changes in the hosts or storage controllers. SRCMap ties together disparate ideas from server and storage energy management (namely caching, replication, transparent live migration, and write off-loading) to minimize the energy drawn by storage devices in a data center. It continuously targets energy proportionality by dynamically increasing or decreasing the number of active physical volumes in a data center in response to variation in I/O workload intensity.

SRCMap is based on the following observations in production workloads detailed in §3.3: *(i)* the active data set in storage volumes is small, *(ii)* this active data set is stable, and *(iii)* there is substantial variation in workload intensity both within and across storage volumes. Thus, instead of creating full replicas of data volumes,

SRCMap creates partial replicas that contain the working sets of data volumes. The small replica size allows creating multiple copies on one or more target volumes or analogously allowing one target volume to host replicas of multiple source volumes. Additional space is reserved on each partial replica to offload writes [NDR08] to volumes that are spun down.

SRCMap enables a high degree of flexibility in spinning down volumes because it activates either the primary volume or exactly one working set replica of each volume at any time. Based on the aggregate workload intensity, SRCMap changes the set of active volumes in the granularity of hours rather than minutes to address the reliability concerns related to the limited number of disk spin-up cycles. It selects active replica targets that allow spinning down the maximum number of volumes, while serving the aggregate storage workload. The virtualization layer remaps the virtual to physical volume mapping as required thereby replacing expensive data migration operations with background data synchronization operations. SRCMap is able to create close to N *power-performance levels on a storage subsystem with* N *volumes*, enabling storage energy consumption proportional to the I/O workload intensity.

In this chapter we propose design goals for energy proportional storage systems (§3.2), analyze storage workload characteristics (§3.3) that motivate design choices (§3.4), provide detailed system design, algorithms, and optimizations (§3.5 and §3.6), and evaluate for energy proportionality (§3.7).

3.2 Design Goals

In this section, we identify the goals for a practical and effective energy proportional storage systems.

1. Fine-grained energy proportionality: Energy proportional storage systems are uniquely characterized by multiple performance-power levels. True energy proportionality requires that for a system with a peak power of P_{peak} for a workload intensity ρ_{max}, the power drawn for a workload intensity ρ_i would be:

$$P_{peak} \times \frac{\rho_i}{\rho_{max}} \tag{3.1}$$

2. Low space overhead: Replication-based strategies could achieve energy proportionality trivially by replicating each volume on all the other $N - 1$ volumes. This would require N copies of each volume, representing an unacceptable space overhead. A practical energy proportional system should incur minimum space overhead; for example, 25% additional space is often available.

3. Reliability: Disk drives are designed to survive a limited number of spin-up cycles [KS07]. Energy conservation based on spinning down the disk must ensure that the additional number of spin-up cycles induced during the disks' expected lifetime is significantly lesser than the manufacturer specified maximum spin-up cycles.

4. Workload shift adaptation: The popularity of data changes, even if slowly over time. Energy management for storage systems that rely on caching popular data over long intervals should address any shift in popularity, while ensuring energy proportionality.

5. Heterogeneity support: A data center is typically composed of several substantially different storage systems (e.g., with variable numbers and types of drives). An ideal energy proportional storage system should account for the differences in their performance-power ratios to provide the best performance at each host level.

Workload Volume	Size [GB]	Reads [GB]		Writes [GB]		Volume accessed
		Total	Uniq	Total	Uniq	
mail	500	62.00	29.24	482.10	4.18	6.27%
homes	470	5.79	2.40	148.86	4.33	1.44%
web-vm	70	3.40	1.27	11.46	0.86	2.8%

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

3.3 Storage Workload Characteristics

In this section, we characterize the nature of I/O access on servers using workloads from three production systems, specifically looking for properties that help us in our goal of energy proportional storage. The systems include an email server (*mail* workload), a virtual machine monitor running two web servers (*web-vm* workload), and a file server (*homes* workload). The *mail* workload serves user INBOXes for the entire Computer Science department at FIU. The *homes* workload is that of a NFS server that serves the home directories for our research group at FIU; activities represent those of a typical researcher consisting of software development, testing, and experimentation, the use of graph-plotting software, and technical document preparation. Finally, the *web-vm* workload is collected from a virtualized system that hosts two CS department web-servers, one hosting the department's online course management system and the other hosting the department's web-based email access portal.

In each system, we collected I/O traces downstream of an active page cache for a duration of three weeks. Average weekly statistics related to these workloads are summarized in Table 3.1. The first thing to note is that the weekly working sets (unique accesses during a week) is a small percentage of the total volume size (1.5-6.5%). This trend is consistent across all volumes and leads to our first observation.

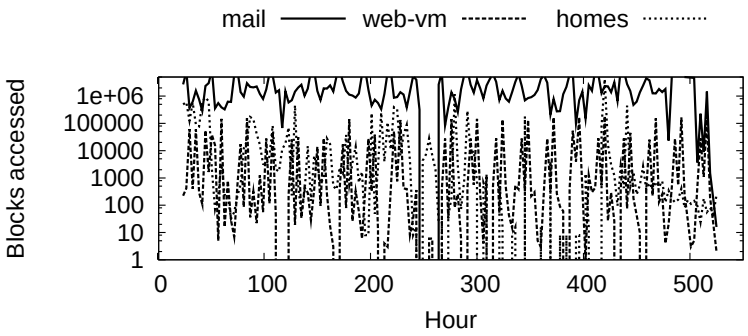


Figure 3.1: Variability in I/O workload intensity

Observation 1. *The active data set for storage volumes is typically a small fraction of total used storage.*

Dynamic consolidation utilizes variability in I/O workload intensity to increase or decrease the number of active devices. Figure 3.1 depicts large variability in I/O workload intensity for each of the three workloads over time, with as much as 5-6 orders of magnitude between the lowest and highest workload intensity levels across time. This highlights the potential of energy savings if the storage systems can be made energy proportional.

Observation 2. *There is a significant variability in I/O workload intensity on storage volumes.*

Based on our first two observations, we hypothesize that there is room for powering down physical volumes that are substantially under-utilized by replicating a small active working-set on other volumes which have the spare bandwidth to serve accesses to the powered down volumes. This motivates *Sample* and *Replicate* in SRCMap. Energy conservation is possible provided the corresponding working set replicas can serve most requests to each powered down volume. This would be true if working sets are largely stable.

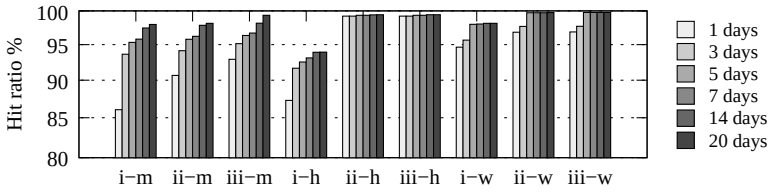


Figure 3.2: Overlap in daily working sets for the mail (m), homes (h), and web-vm (w) workloads.(i) Reads and writes against working set , (ii) Reads against working set and (iii) Reads against working set, recently offloaded writes, and recent missed reads.

We investigate the stability of the volume working sets in Fig. 3.2 for three progressive definitions of the working set. In the first scenario, we compute the classical working set based on the last few days of access history. In the second scenario, we additionally assume that writes can be offloaded and mark all writes as hits. In the third scenario, we further expand the working set to include recent writes and past missed reads. For each scenario, we compute the working set hits and misses for the following day's workload and study the hit ratio with change in the length of history used to compute the working set. We observe that the hit ratio progressively increases both across the scenarios and as we increase the history length leading us to conclude that data usage exhibits high temporal locality and that the working set after including recent accesses is fairly stable. This leads to our third observation (also observed earlier by Leung *et al.* [LPGM08]).

Observation 3. *Data usage is highly skewed with more than* 99% *of the working set consisting of some 'really popular' data and 'recently accessed' data.*

The first three observations are the pillars behind the *Sample*, *Replicate* and *Consolidate* approach whereby we sample each volume for its working set, replicate these working sets on other volumes, and consolidate I/O workloads on proportionately fewer volumes during periods of low load. Before designing a new system based on

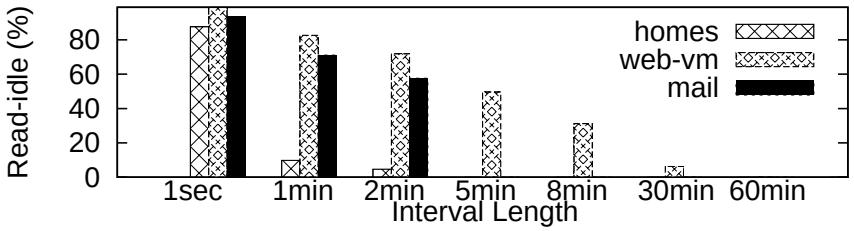


Figure 3.3: Distribution of read-idle times

the above observations, we study the suitability of a simpler *write-offloading* technique for building energy proportional storage systems. Write off-loading is based on the observation that I/O workloads are write dominated and simply off-loading writes to a different volume can cause volumes to be idle for a substantial fraction (79% for workloads in the original study) of time [NDR08]. While write off-loading increases the fraction of idle time of volumes, the distribution of idle time durations due to write off-loading raises an orthogonal, but important, concern. If these idle time durations are short, saving energy requires frequent spinning down/up of the volumes which degrades reliability of the disk drives.

Figure 3.3 depicts the read-idle time distributions of the three workloads. It is interesting to note that idle time durations for the *homes* and *mail* workloads are all less than or equal to 2 minutes, and for the *web-vm* the majority are less than or equal to 5 minutes are all are less than 30 minutes.

Observation 4. *The read-idle time distribution (periods of writes alone with no intervening read operations) of I/O workloads is dominated by small durations, typically less than five minutes.*

This observation implies that exploiting all read-idleness for saving energy will necessitate spinning up the disk at least 720 times a day in the case of *homes* and *mail* and at least 48 times in the case of *web-vm*. This can be a significant hurdle

to reliability of the disk drives which typically have limited spin-up cycles [KS07]. It is therefore important to develop new techniques that can substantially increase average read-idle time durations.

3.4 Background and Rationale

Storage virtualization managers simplify storage management by enabling a uniform view of disparate storage resources in a data center. They export a storage controller interface allowing users to create logical volumes or virtual disks (*vdisks*) and mount these on hosts. The physical volumes managed by the physical storage controllers are available to the virtualization manager as managed disks (*mdisks*) entirely transparently to the hosts which only view the logical *vdisk* volumes. A useful property of the virtualization layer is the complete flexibility in allocation of *mdisk* extents to *vdisk*s.

Applying server consolidation principles to storage consolidation using virtualization would activate only the most energy-efficient *mdisk*s required to serve the aggregate workload during any period T. Data from the other *mdisk*s chosen to be spun down would first need to be migrated to active *mdisk*s to effect the change. While data migration is an expensive operation, the ease with which virtual-to-physical mappings can be reconfigured provides an alternative approach. A naïve strategy following this approach could replicate data for each *vdisk* on all the *mdisk*s and adapt to workload variations by dynamically changing the virtual-to-physical mappings to use only the selected *mdisk*s during T. Unfortunately, this strategy requires N times additional space for a N *vdisk* storage system, an unacceptable space overhead.

SRCMap intelligently uses the storage virtualization layer as an I/O indirection mechanism to deliver a practically feasible, energy proportional solution. Since it op-

erates at the storage virtualization manager, it does not alter the basic redundancy-based reliability properties of the underlying physical volumes which is determined by the respective physical volume (e.g., RAID) controllers. To maintain the redundancy level, SRCMap ensures that a volume is replicated on target volumes at the same RAID level. While we detail SRCMap's design and algorithms in subsequent sections (§ 3.5 and § 3.6), here we list the rationale behind SRCMap's design decisions. These design decisions together help to satisfy the design goals for an ideal energy proportional storage system.

I. Multiple replica targets. Fine-grained energy proportionality requires the flexibility to increase or decrease the number of active physical volumes one at a time. Techniques that activate a fixed secondary device for each data volume during periods of low activity cannot provide the flexibility necessary to deactivate an arbitrary fraction of the physical volumes. In SRCMap, we achieve this fine-grained control by creating a primary *mdisk* for each *vdisk* and replicating only the working set of each *vdisk* on multiple secondary *mdisk*s. This ensures that *(a)* every volume can be offloaded to one of multiple targets and *(b)* each target can serve the I/O workload for multiple *vdisk*s. During peak load, each *vdisk* maps to its primary *mdisk* and all *mdisk*s are active. However, during periods of low activity, SRCMap selects a proportionately small subset of *mdisk*s that can support the aggregate I/O workload for all *vdisk*s.

II. Sampling. Creating multiple full replicas of *vdisk*s is impractical. Drawing from *Observation 1* (§ 3.3), SRCMap substantially reduces the space overhead of maintaining multiple replicas by sampling only the working set for each *vdisk* and replicating it. Since the working set is typically small, the space overhead is low.

III. Ordered replica placement. While sampling helps to reduce replica sizes substantially, creating multiple replicas for each sample still induces space overhead.

In SRCMap, we observe that all replicas are not created equal; for instance, it is more beneficial to replicate a lightly loaded volume than a heavily loaded one which is likely to be active anyway. Similarly, a large working set has greater space overhead; SRCMap chooses to create fewer replicas aiming to keep it active, if possible. As we shall formally demonstrate, carefully ordering the replica placement helps to minimize the number of active disks for fine-grained energy proportionality.

IV. Dynamic source-to-target mapping and dual data synchronization. From *Observation 2* (§ 3.3), we know that workloads can vary substantially over a period of time. Hence, it is not possible to pre-determine which volumes need to be active. Target replica selection for any volume being powered down therefore needs to be a dynamic decision and also needs to take into account that some volumes have more replicas (or target choices) than others. We use two distinct mechanisms for updating the replica working sets. The active replica lies in the data path and is immediately synchronized in the case of a *read miss*. This ensures that the active replica continuously *adapts* with change in *workload popularity*. The secondary replicas, on the other hand, use a lazy, incremental data synchronization in the background between the primary replica and any secondary replicas present on active *mdisk*s. This ensures that switching between replicas requires minimal data copying and can be performed fairly quickly.

V. Coarse-grained power cycling. In contrast to most existing solutions that rely on fine-grained disk power-mode switching, SRCMap implements coarse-grained consolidation intervals (of the order of hours), during each of which the set of active *mdisk*s chosen by SRCMap does not change. This ensures normal disk lifetimes are realized by adhering to the disk power cycle specification contained within manufacturer data sheets.

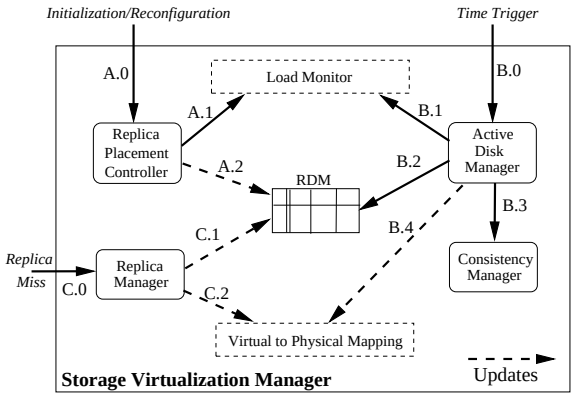


Figure 3.4: SRCMap integrated into a Storage Virtualization Manager. Arrows depict control flow. Dashed/solid boxes denote existing/new components.

3.5 Design Overview

SRCMap is built in a modular fashion to directly interface with storage virtualization managers or be integrated into one as shown in Figure 3.4. The overall architecture supports the following distinct flows of control:

(i) the *replica generation flow* (Flow A) identifies the working set for each *vdisk* and replicates it on multiple *mdisk*s. This flow is orchestrated by the *Replica Placement Controller* and is triggered once when SRCMap is initialized and whenever a configuration change (e.g., addition of a new workload or new disks) takes place. Once a trigger is generated, the *Replica Placement Controller* obtains a historical workload trace from the *Load Monitor* and computes the working set and the long-term workload intensity for each volume (*vdisk*). The working set is then replicated on one or more physical volumes (*mdisks*). The blocks that constitute the working set for the *vdisk* and the target physical volumes where these are replicated are managed using a common data structure called the *Replica Disk Map (RDM).*

(ii) the *active disk identification flow* (Flow B) identifies, for a period T, the active *mdisk*s and activated replicas for each inactive *mdisk*. The flow is triggered at the

beginning of the consolidation interval T (e.g., every 2 hours) and orchestrated by the *Active Disk Manager*. In this flow, the *Active Disk Manager* queries the *Load Monitor* for expected workload intensity of each *vdisk* in the period T. It then uses the workload information along with the placement of working set replicas on target *mdisks* to compute the set of active primary *mdisk*s and a active secondary replica *mdisk* for each inactive primary *mdisk*. It then directs the *Consistency Manager* to ensure that the data on any selected active primary or active secondary replica is current. Once consistency checks are made, it updates the *Virtual to Physical Mapping* to redirect the workload to the appropriate *mdisk*.

(iii) the *I/O redirection flow* (Flow C) is an extension of the I/O processing in the *storage virtualization manager* and utilizes the built-in virtual-to-physical remapping support to direct requests to primaries or active replicas. Further, this flow ensures that the working-set of each *vdisk* is kept up-to-date. To ensure this, whenever a request to a block not available in the active replica is made, a *Replica Miss* event is generated. On a *Replica Miss*, the *Replica Manager* spin-ups the primary *mdisk* to fetch the required block. Further, it adds this new block to the working set of the *vdisk* in the RDM. We next describe the key components of SRCMap.

3.5.1 Load Monitor

The *Load Monitor* resides in the storage virtualization manager and records access to data on any of the *vdisk*s exported by the virtualization layer. It provides two interfaces for use by SRCMap – long-term workload data interface invoked by the *Replica Placement Controller* and predicted short-term workload data interface invoked by the *Active Disk Manager*.

3.5.2 Replica Placement Controller

The *Replica Placement Controller* orchestrates the process of *Sampling* (identifying working sets for each *vdisk*) and *Replicating* on one or more target *mdisk*s. We use a conservative definition of working set that includes all the blocks that were accessed during a fixed duration, configured as the minimum duration beyond which the hit ratio on the working set saturates. Consequently, we use 20 days for *mail*, 14 days for *homes* and 5 days for *web-vm* workload (Fig. 3.2). The blocks that capture the working set for each *vdisk* and the *mdisks* where it is replicated are stored in the RDM. The details of the parameters and methodology used within *Replica Placement* are described in Section 3.6.1.

3.5.3 Active Disk Manager

The *Active Disk Manager* orchestrates the *Consolidate* step in SRCMap. The module takes as input the workload intensity for each *vdisk* and identifies if the primary *mdisk* can be spun down by redirecting the workload to one of the secondary *mdisk*s hosting its replica. Once the target set of active *mdisk*s and replicas are identified, the *Active Disk Manager* synchronizes the identified active primaries or active secondary replicas and updates the virtual-to-physical mapping of the storage virtualization manager, so that I/O requests to a *vdisk* could be redirected accordingly. The Active Disk Manager uses a *Consistency Manager* for the synchronization operation. Details of the algorithm used by *Active Disk Manager* for selecting active *mdisks* are described in Section 3.6.2.

3.5.4 Consistency Manager

The *Consistency Manager* ensures that the primary *mdisk* and the replicas are consistent. Before an *mdisk* is spun down and a new replica activated, the new active replica is made consistent with the previous one. In order to ensure that the overhead during the re-synchronization is minimal, an incremental point-in-time (PIT) relationship (e.g., Flash-copy in IBM SVC [IBM]) is maintained between the active data (either the primary *mdisk* or one of the active replicas) and all other copies of the same data. A *go-to-sync* operation is performed periodically between the active data and all its copies on active *mdisk*s. This ensures that when an *mdisk* is spun up or down, the amount of data to be synchronized is small.

3.5.5 Replica Manager

The *Replica Manager* ensures that the replica data set for a *vdisk* is able to mimic the working set of the *vdisk* over time. If a data block unavailable at the active replica of the *vdisk* is read causing a *replica miss*, the Replica Manager copies the block to the replica space assigned to the active replica and adds the block to the *Replica Metadata* accordingly. Finally, the *Replica Manager* uses a Least Recently Used (LRU) policy to evict an older block in case the replica space assigned to a replica is filled up. If the active data set changes drastically, there may be a large number of *replica misses*. All these replica misses can be handled by a single spin-up of the primary *mdisk*. Once all the data in the new working set is touched, the primary *mdisk* can be spun-down as the active replica is now up-to-date. The continuous updating of the *Replica Metadata* enables SRCMap to meet the goal of *Workload shift adaptation*, without re-running the expensive *replica generation flow*.

The *replica generation flow* needs to re-run only when a disruptive change occurs such as addition of a new workload or a new volume or new disks to a volume.

3.6 Algorithms and Optimizations

In this section, we present details about the algorithms employed by SRCMap. We first present the long-term replica placement methodology and subsequently, the short-term active disk identification method.

3.6.1 Replica Placement Algorithm

The *Replica Placement Controller* creates one or more replicas of the working set of each *vdisk* on the available replica space on the target *mdisk*s. We use the insight that all replicas are not created equal and have distinct associated costs and benefits. The space cost of creating the replica is lower if the *vdisk* has a smaller working set. Similarly, the benefit of creating a replica is higher if the *vdisk* (i) has a stable working set (lower misses if the primary *mdisk* is switched off), (ii) has a small average load making it easy to find spare bandwidth for it on any target *mdisk*, and (iii) is hosted on a less power-efficient primary *mdisk*. Hence, the goal of both *Replica Placement* and *Active Disk Identification* is to ensure that we create more replicas for *vdisk*s that have a favorable cost-benefit ratio. The goal of the replica placement is to ensure that if the *Active Disk Manager* decides to spin down the primary *mdisk* of a *vdisk*, it should be able to find at least one active target *mdisk* that hosts its replica, captured in the following *Ordering Property*.

Definition 1. Ordering Property*: For any two vdisks V_i and V_j, if V_i is more likely to require a replica target than V_j at any time t during* Active Disk Identification*, then V_i is more likely than V_j to find a replica target amongst active mdisks at time t.*

The *replica placement algorithm* consists of (i) creating an initial ordering of *vdisk*s in terms of cost-benefit tradeoff (ii) a bipartite graph creation that reflects this ordering (iii) iteratively creating one source-target mapping respecting the current order and (iv) re-calibration of edge weights to ensure the *Ordering Property* holds for the next iteration of source-target mapping.

Initial *vdisk* ordering The Initial *vdisk* ordering creates a sorted order amongst *vdisk*s based on their cost-benefit tradeoff. For each *vdisk* V_i, we compute the probability P_i that its primary *mdisk* M_i would be spun down as

$$P_i = \frac{w_1 WS_{min}}{WS_i} + \frac{w_2 PPR_{min}}{PPR_i} + \frac{w_3 \rho_{min}}{\rho_i} + \frac{w_f m_{min}}{m_i} \tag{3.2}$$

where the w_k are tunable weights, WS_i is the size of the working set of V_i, PPR_i is the performance-power ratio (ratio between the peak IO bandwidth and peak power) for the primary mdisk M_i of V_i, ρ_i is the average long-term I/O workload intensity (measured in IOPS) for V_i, and m_i is the number of read misses in the working set of V_i, normalized by the number of spindles used by its primary mdisk M_i. The corresponding *min* subscript terms represent the minimum values across all the *vdisk*s and provide normalization. The probability formulation is based on the dual rationale that it is relatively easier to find a target *mdisk* for a smaller workload and switching off relatively more power-hungry disks saves more energy. Further, we assign a higher probability for spinning down *mdisk*s that host more stable working sets by accounting for the number of times a read request cannot be served from the replicated working set, thereby necessitating the spinning up of the primary *mdisk*.

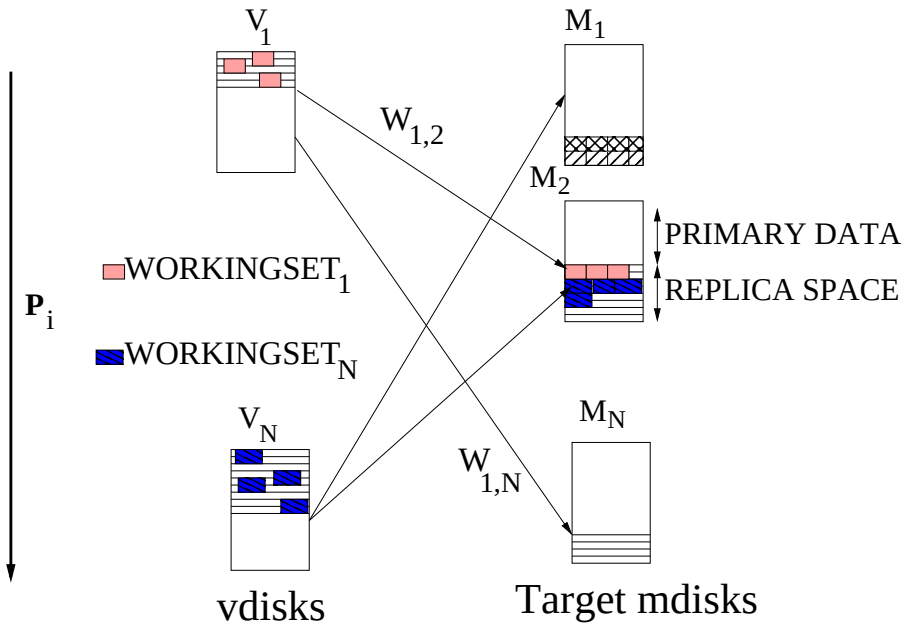


Figure 3.5: Replica Placement Model

Bipartite graph creation *Replica Placement* creates a bipartite graph $G(V \rightarrow M)$ with each *vdisk* as a source node V_i, its primary *mdisk* as a target node M_i, and the edge weights $e(V_i, M_j)$ representing the cost-benefit trade-off of placing a replica of V_i on M_j (Fig. 3.5). The nodes in the bipartite graph are sorted using P_i (disks with larger P_i are at the top). We initialize the edge weights $w_{i,j} = P_i$ for each edge $e(V_i, M_j)$ (source-target pair). Initially, there are no replica assignments made to any target *mdisk*. The replica placement algorithm iterates through the following two steps, until all the available replica space on the target *mdisk*s have been assigned to source *vdisk* replicas. In each iteration, exactly one target *mdisk*'s replica space is assigned.

Source-Target mapping The goal of the replica placement method is to achieve a source target mapping that achieves the *Ordering property*. To achieve this goal,

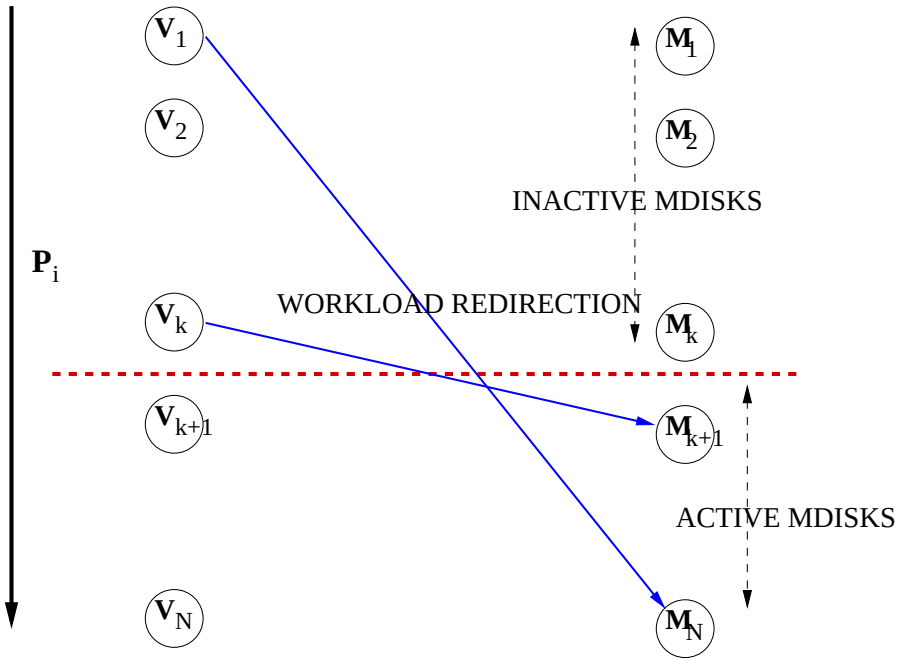


Figure 3.6: Active Disk Identification

the algorithm takes the top-most target *mdisk* M_i whose replica space is not yet assigned and selects the set of highest weight incident edges such that the combined replica size of the source nodes in this set fills up the replica space available in M_i (e.g, the working sets of V_1 and V_N are replicated in the replica space of M_2 in Fig. 3.5). When the replica space on a target *mdisk* is filled up, we mark the target *mdisk* as assigned. One may observe that this procedure always gives preference to source nodes with a larger P_i. Once an *mdisk* finds a replica, the likelihood of it requiring another replica decreases and we factor this using a re-calibration of edge weights, which is detailed next.

Re-calibration of edge weights We observe that the initial assignments of weights ensure the *Ordering property*. However, once the working set of a *vdisk* V_i has been replicated on a set of target *mdisk*s $\mathbf{T_i} = M_1, \ldots, M_{least}$ (M_{least} is the

mdisk with the least P_i in $\mathbf{T_i}$) s.t. $P_i > P_{least}$, the probability that V_i would require a new target *mdisk* during *Active Disk Identification* is the probability that both M_i and M_{least} would be spun down. Hence, to preserve the *Ordering property*, we re-calibrate the edge weights of all outgoing edges of any primary *mdisk*s S_i assigned to target mdisks T_j as

$$\forall k \quad w_{i,k} = P_j P_i \tag{3.3}$$

Once the weights are recomputed, we iterate from the Source-Target mapping step until all the replicas have been assigned to target *mdisk*s. One may observe that the re-calibration succeeds in achieving the *Ordering* property because we start assigning the replica space for the top-most target *mdisk*s first. This allows us to increase the weights of source nodes monotonically as we place more replicas of its working set. We formally prove the following result in the appendix.

Theorem 1. *The* Replica Placement Algorithm *ensures ordering property.*

3.6.2 Active Disk Identification

We now describe the methodology employed to identify the set of active *mdisk*s and replicas at any given time. For ease of exposition, we define the probability P_i of a primary *mdisk* M_i equal to the probability P_i of its *vdisk* V_i. Active disk identification consists of:

I: Active mdisk Selection: We first estimate the expected aggregate workload to the storage subsystem in the next interval. We use the workload to a *vdisk* in the previous interval as the predicted workload in the next interval for the *vdisk*. The aggregate workload is then estimated as sum of the predicted workloads for all *vdisk*s in the storage system. This aggregate workload is then used to identify the minimum subset of *mdisk*s (ordered by reverse of P_i) such that the aggregate bandwidth of these *mdisk*s exceeds the expected aggregate load.

II: Active Replica Identification: This step elaborated shortly identifies one (of the many possible) replicas on an active *mdisk* for each inactive *mdisk* to serve the workload redirected from the inactive *mdisk*.

III: Iterate: If the Active Replica Identification step succeeds in finding an active replica for all the inactive *mdisk*s, the algorithm terminates. Else, the number of active *mdisk*s are increased by 1 and the algorithm repeats the Active Replica Identification step.

One may note that since the number of active disks are based on the maximum predicted load in a consolidation interval, a sudden increase in load may lead to an increase in response times. If performance degradation beyond user-defined acceptable levels persists beyond a user-defined interval (e.g, 5 mins), the *Active Disk Identification* is repeated for the new load.

S = set of disks to be spun down
A = set of disks to be active
Sort S by reverse of P_i
Sort A by P_i
For each $D_i \in S$
  For each $D_j \in A$
    If D_j hosts a replica R_i of D_i AND
    D_j has spare bandwidth for R_i
     $Candidate(D_i) = D_j$, $break$
  End-For
  If $Candidate(D_i)$==$null$ return $Failure$
End-for
$\forall i$, $D_i \in S$ return $Candidate(D_i)$

Algorithm 1: Active Replica Identification algorithm

Active Replica Identification Fig. 3.6 depicts the high-level goal of Active Replica Identification, which is to have the primary *mdisk*s for *vdisk*s with larger P_i spun down, and their workload directed to few *mdisk*s with smaller P_i. To do so, it must identify an active replica for each inactive primary *mdisk*, on one of the

active *mdisk*s. The algorithm uses two insights: (i) The *Replica Placement* process creates more replicas for *vdisk*s with a higher probability of being spun down (P_i) and (ii) primary *mdisk*s with larger P_i are likely to be spun down for a longer time.

To utilize the first insight, we first allow primary *mdisk*s with small P_i, which are marked as inactive, to find an active replica, as they have fewer choices available. To utilize the second insight, we force inactive primary *mdisk*s with large P_i to use a replica on active *mdisk*s with small P_i. For example in Fig. 3.6, *vdisk* V_k has the first choice of finding an active *mdisk* that hosts its replica and in this case, it is able to select the first active *mdisk* M_{k+1}. As a result, inactive *mdisk*s with larger P_i are mapped to active *mdisk*s with the smaller P_i (e.g, V_1 is mapped to M_N). Since an *mdisk* with the smallest P_i is likely to remain active most of the time, this ensures that there is little to no need to 'switch active replicas' frequently for the inactive disks. The details of this methodology are described in Fig. 1.

3.6.3 Key Optimizations to Basic SRCMap

We augment the basic SRCMap algorithm to increase its practical usability and effectiveness as follows.

Sub-volume creation SRCMap redirects the workload for any primary *mdisk* that is spun down to exactly one target *mdisk*. Hence, a target *mdisk* M_j for a primary *mdisk* M_i needs to support the combined load of the vdisks V_i and V_j in order to be selected. With this requirement, the SRCMap consolidation process may incur a fragmentation of the available I/O bandwidth across all volumes. To elaborate, consider an example scenario with 10 identical *mdisk*s, each with capacity C and input load of $C/2 + \delta$. Note that even though this load can be served using $10/2 + 1$ *mdisk*s, there is no single *mdisk* can support the input load of 2 *vdisk*s.

To avoid such a scenario, SRCMap sub-divides each *mdisk* into N_{SV} sub-volumes and identifies the working set for each sub-volume separately. The sub-replicas (working sets of a sub-volume) are then placed independently of each other on target *mdisk*s. With this optimization, SRCMap is able to subdivide the least amount of load that can be migrated, thereby dealing with the fragmentation problem in a straightforward manner.

This optimization requires a complementary modification to the*Replica Placement* algorithm. The Source-Target mapping step is modified to ensure that sub-replicas belonging to the same source *vdisk* are not co-located on a target *mdisk*.

Scratch Space for Writes and Missed Reads SRCMap incorporates the basic write off-loading mechanism as proposed by Narayanan *et al.* [NDR08]. The current implementation of SRCMap uses an additional allocation of write scratch space with each sub-replica to absorb new writes to the corresponding portion of the data volume. A future optimization is to use a single write scratch space within each target *mdisk* rather than one per sub-replica within the target *mdisk* so that the overhead for absorbing writes can be minimized.

A key difference from write off-loading, however, is that on a *read miss* for a spun down volume, SRCMap additionally offloads the data read to dynamically learn the working-set. This helps SRCMap achieve the goal of *Workload Shift Adaptation* with change in working set. While write off-loading uses the inter read-miss durations exclusively for spin down operations, SRCMap targets capturing entire working-sets including both reads and writes in replica locations to prolong read-miss durations to the order of hours and thus places more importance on learning changes in the working-set.

3.7 Evaluation

In this section, we evaluate SRCMap using a prototype implementation of SRCMap-based storage virtualization manager and an energy simulator seeded by the prototype. We investigate the following questions:

1. What degree of proportionality in energy consumption and I/O load can be achieved using SRCMap?
2. How does SRCMap impact reliability?
3. What is the impact of storage consolidation on the I/O performance?
4. How sensitive are the energy savings to the amount of over-provisioned space?
5. What is the overhead associated with implementing an SRCMap indirection optimization?

Workload The workloads used consist of I/O requests to eight independent data volumes, each mapped to an independent disk drive. In practice, volumes will likely comprise of more than one disk, but resource restrictions did not allow us to create a more expansive testbed. We argue that *relative* energy consumption results still hold despite this approximation. These volumes support a mix of production web-servers from the FIU CS department data center, end-user *homes* data, and our lab's Subversion (SVN) and Wiki servers as detailed in Table 3.2.

Workload I/O statistics were obtained by running *blktrace* [ABO07] on each volume. Observe that there is a wide variance in their load intensity values, creating opportunities for consolidation across volumes.

Storage Testbed For experimental evaluation, we set up a single machine (Intel Pentium 4 HT 3GHz, 1GB memory) connected to 8 disks via two SATA-II controllers A and B. The cumulative (merged workload) trace is played back using

Volume	ID	Disk Model	Size [GB]	Avg IOPS	Max IOPS
home-1	D0	WD5000AAKB	270	8.17	23
online	D1	WD360GD	7.8	22.62	82
webmail	D2	WD360GD	7.8	25.35	90
webresrc	D3	WD360GD	10	7.99	59
webusers	D4	WD360GD	10	18.75	37
svn-wiki	D5	WD360GD	20	1.12	4
home-2	D6	WD2500AAKS	170	0.86	4
home-3	D7	WD2500AAKS	170	1.37	12

Table 3.2: Workload and storage system details.

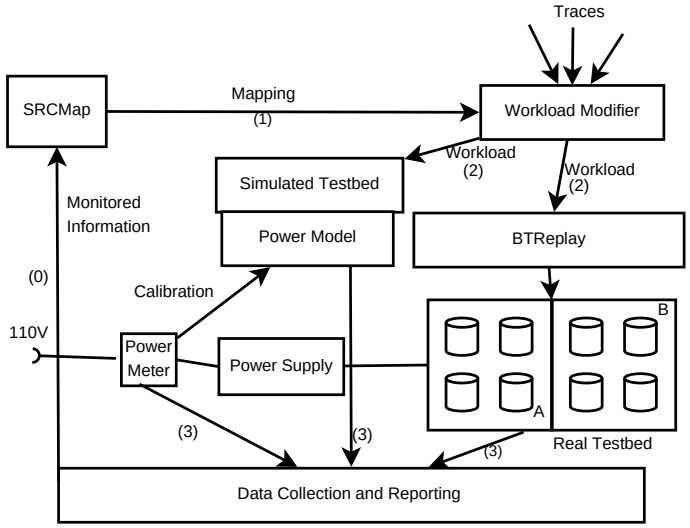


Figure 3.7: Logical view of experimental setup

btreplay [ABO07] with each volume's trace played back to the corresponding disk. All the disks share one power supply P that is dedicated only for the experimental drives; the machine connects to another power supply. The power supply P is connected to a *Watts up? PRO* power meter [Wat09] which allows us to measure power consumption at a one second granularity with a resolution of 0.1W. An overhead of 6.4W is introduced by the power supply itself which we deduct from all our power measurements.

Experimental Setup We describe the experimental setup used in our evaluation study in Fig. 3.7. We implemented an SRCMap module with its algorithms for

replica placement and active disk identification during any consolidation interval. An overall experimental run consists of using the monitored data to (1) identify the consolidation candidates for each interval and create the virtual-to-physical mapping (2) modify the original traces to reflect the mapping and replaying it, and (3) power and response time reporting. At each consolidation event, the *Workload Modifier* generates the necessary additional I/O to synchronize data across the sub-volumes affected due to active replica changes.

We evaluate SRCMap using two different sets of experiments: *(i)* prototype runs and *(ii)* simulated runs. The prototype runs evaluate SRCMap against a real storage system and enable realistic measurements of power consumption and impact to I/O performance via the reporting module. In a prototype run, the modified I/O workload is replayed on the actual testbed using *btreplay* [ABO07].

The simulator runs operate similarly on a simulated testbed, wherein a power model instantiated with power measurements from the testbed is used for reporting the power numbers. The advantage with the simulator is the ability to carry out longer duration experiments in simulated time as opposed to real-time allowing us to explore the parameter space efficiently. Further, one may use it to simulate various types of storage testbeds to study the performance under various load conditions. In particular, we use the simulator runs to evaluate energy-proportionality by simulating the testbed with different values of disk IOPS capacity estimates. We also simulate alternate power management techniques (e.g., caching, replication) for a comparative evaluation.

All experiments with the prototype and the simulator were performed with the following configuration parameters. The consolidation interval was chosen to be 2 hours for all experiments to restrict the worst-case spin-up cycles for the disk drives to an acceptable value. Two minute disk timeouts were used for inactive disks; active

Volume ID	L(0) [IOPS]	L(1) [IOPS]	L(2) [IOPS]	L(3) [IOPS]	L(4) [IOPS]
D0	33	57	74	96	125
D1-D5	52	89	116	150	196
D6, D7	38	66	86	112	145

(a)

0	1	2	3	4	5	6	7	8
19.8	27.2	32.7	39.1	44.3	49.3	55.7	59.7	66.1

(b)

Table 3.3: Experimental settings: (a) Estimated disk IOPS capacity levels. (b) Storage system power consumption in Watts as the number of disks in active mode are varied from 0 to 8.All disks consumed approximately the same power when active. The disks not in active mode consume standby power which was found to be the same across all disks.

disks within a consolidation interval remain continuously active. Working sets and replicas were created based on a three week workload history and we report results for a subsequent 24 hour duration for brevity. The consolidation is based on an estimate of the disk IOPS capacity, which varies for each volume. We computed an estimate of the disk IOPS using a synthetic random I/O workload for each volume separately (Level $L1$). We use 5 IOPS estimation levels (L0 through L4) to (a) simulate storage testbeds at different load factors and (b) study the sensitivity of SRCMap with the volume IOPS estimation. The per volume sustainable IOPS at each of these load levels is provided in Table 3.3(a). The power consumption of the storage system with varying number of disks in active mode is presented in Table 3.3(b).

3.7.1 Prototype Results

For the prototype evaluation, we took the most dynamic 8-hour period (4 consolidation intervals) from the 24 hours and played back I/O traces for the 8 workloads described earlier in real-time. We report actual power consumption and the I/O response time (which includes queuing and service time) distribution for SRCMap

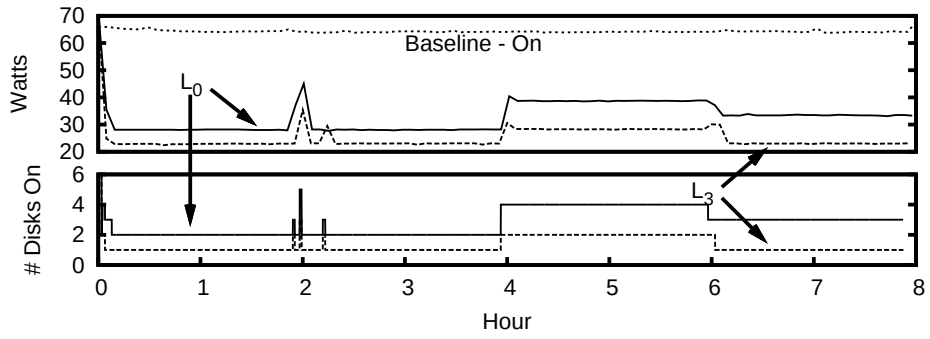


Figure 3.8: Power and active disks time-line.

when compared to a baseline configuration where all disks are continuously active. Power consumption was measured every second and disk active/standby state information was polled every 5 seconds. We used 2 different IOPS levels; $L0$ when a very conservative (low) estimate of the disk IOPS capacity is made and $L3$ when a reasonably aggressive (high) estimate is made.

We study the power savings due to SRCMap in Figure 3.8. Even using a conservative estimate of disk IOPS, we are able to spin down approximately 4.33 disks on an average, leading to an average savings of $23.5W$ (35.5%). Using an aggressive estimate of disk IOPS, SRCMap is able to spin down 7 disks saving $38.9W$ (59%) for all periods other than the 4hr-6hr period. In the 4-6 hr period, it uses 2 disks leading to a power savings of $33.4W$ (50%). The spikes in the power consumption relate to planned and unplanned (due to read misses) volume activations, which are few in number. It is important to note that substantial power is used in maintaining standby states ($19.8W$) and within the dynamic range, the power savings due to SRCMap are even higher.

We next investigate any performance penalty incurred due to consolidation. Fig. 3.9 (upper) depicts the cumulative probability density function (CDF) of response times for three different configurations: *Baseline - On* – no consolidation

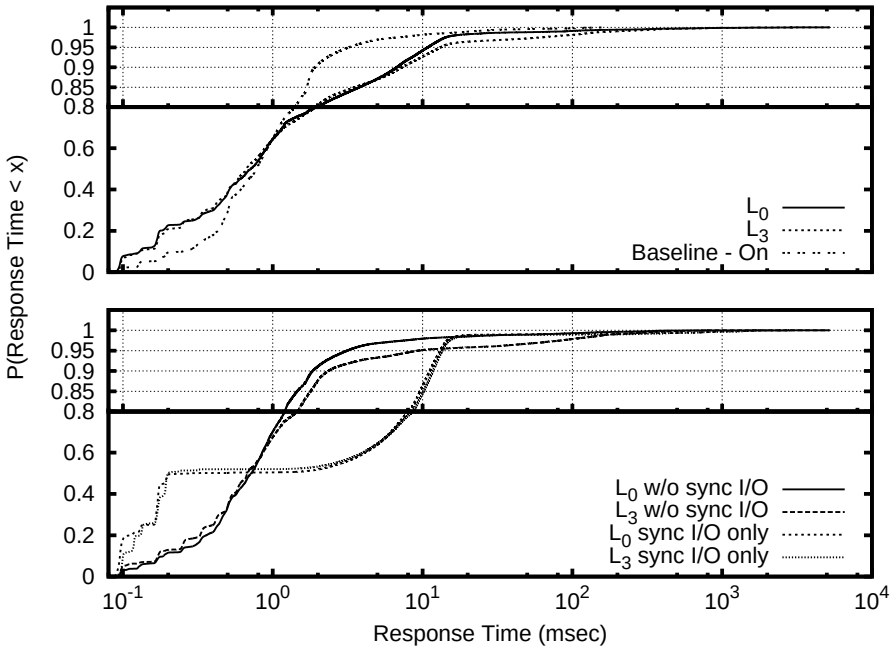


Figure 3.9: Impact of consolidation on response time.

and all disks always active, SRCMap using L_0, and L_3. The accuracy of the CDFs for L_0 and L_3 suffer from a reporting artifact that the CDFs include the latencies for the synchronization I/Os themselves which we were not able to filter out. We throttle the synchronization I/Os to one every 10ms to reduce their interference with foreground operations.

First, we observed that less than 0.003% of the requests incurred a spin-up hit due to read misses resulting in latencies of greater than 4 seconds in both the L_0 and L_3 configurations (not shown). This implies that the working-set dynamically updated with missed reads and offloaded writes is a fairly at capturing the active data for these workloads. Second, we observe that for response times greater than 1ms, *Baseline - On* demonstrates better performance than L_0 and L_3 (upper plot). For both L_0 and L_3, less than 8% of requests incur latencies greater than 10ms, less than 2% of requests incur latencies greater than 100ms. L_0, having more disks at its disposal, shows slightly better response times than L_3. For response times lower

than 1ms a reverse trend is observed wherein the SRCMap configurations do better than *Baseline - On* . We conjectured that this is due to the influence of the low latency writes during synchronization operations.

To further delineate the influence of synchronization I/Os, we performed two additional runs. In the first run, we disable all synchronization I/Os and in the second, we disable all foreground I/Os (lower plot). The CDFs of only the synchronization operations, which show a bimodal distribution with 50% low-latency writes absorbed by the disk buffer and 50% reads with latencies greater than 1.5ms, indicate that synchronization reads are contributing towards the increased latencies in L_0 and L_3 for the upper plot. The CDF without synchronization ('w/o synch') is much closer to *Baseline - On* with a decrease of approximately 10% in the number of request with latencies greater than 1ms. Intelligent scheduling of synchronization I/Os is an important area of future work to further reduce the impact on foreground I/O operations.

3.7.2 Simulator Results

We conducted several experiments with simulated testbeds hosting disks of capacities $L0$ to $L4$. For brevity, we report our observations for disk capacity levels $L0$ and $L3$, expanding to other levels only when required.

Comparative Evaluation We first demonstrate the basic energy proportionality achieved by SRCMap in its most conservative configuration (L0) and three alternate solutions, *Caching-1*, *Caching-2*, and *Replication*. *Caching-1* is a scheme that uses 1 additional physical volume as a cache. If the aggregate load observed is less than the IOPS capacity of the cache volume, the workload is redirected to the cache volume. If the load is higher, the original physical volumes are used. *Caching-2* uses 2 cache

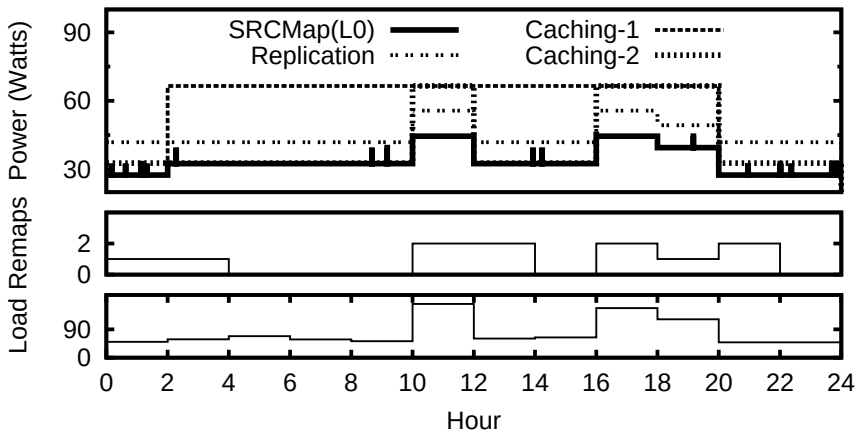


Figure 3.10: Power consumption, remap operations, and aggregate load across time for a single day.

volumes in a similar manner. *Replication* identifies pairs of physical volumes with similar bandwidths and creates replica pairs, where all the data on one volume is replicated on the other. If the aggregate load to a pair is less than the IOPS capacity of one volume, only one in the pair is kept active, else both volumes are kept active.

Figure 3.10 evaluates power consumption of all four solutions by simulating the power consumed as volumes are spun up/down over 12 2-hour consolidation intervals. It also presents the average load (measured in IOPS) within each consolidation interval. In the case of SRCMap, read misses are indicated by instantaneous power spikes which require activating an additional disk drive. To avoid clutter, we do not show the spikes due to read misses for the Cache-1/2 configurations. We observe that each of solutions demonstrate varying degrees of energy proportionality across the intervals. SRCMap (L0) uniformly consumes the least amount of power across all intervals and its power consumption is proportional to load. Replication also demonstrates good energy proportionality but at a higher power consumption on an average. The caching configurations are the least energy proportional with only two effective energy levels to work with.

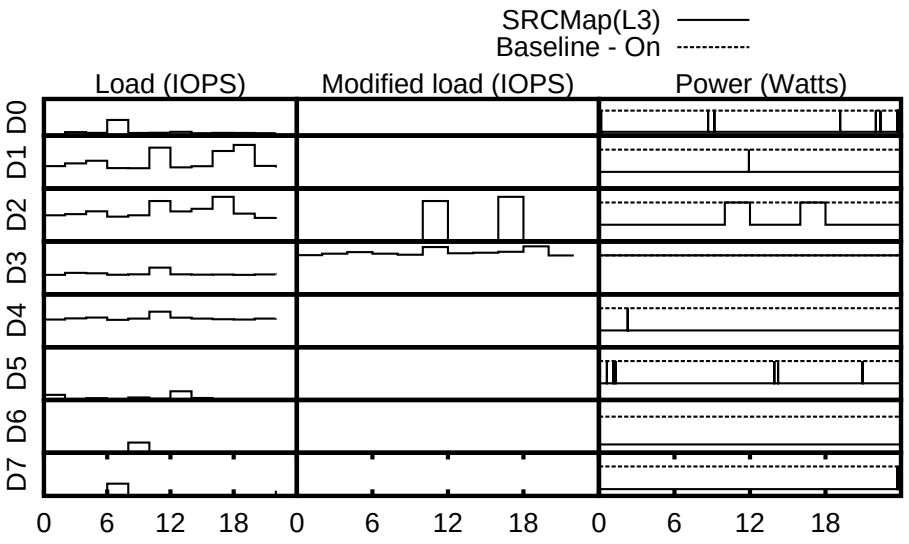


Figure 3.11: Load and power consumption for each disk.. Y ranges for all loads is $[1:130]$ IOPS in logarithmic scale. Y ranges for power is $[0:19]$ W.

We also observe that SRCMap remaps (i.e., changes the active replica for) a minimal number of volumes – either 0, 1, or 2 during each consolidation interval. In fact, we found that for all durations the number of volumes being remapped equaled the change in the number of active physical volumes. indicating that the number of synchronization operations are kept to the minimum. Finally, in our system with eight volumes, Caching-1, Caching-2, and Replication use 12.5%, 25% and 100% additional space respectively, while as we shall show later, SRCMap is able to deliver almost all its energy savings with just 10% additional space.

Next, we investigate how SRCMap modifies per-volume activity and power consumption with an aggressive configuration L3, a configuration that demonstrated interesting consolidation dynamics over the 12 2-hour consolidation intervals. Each row in Figure 3.11 is specific to one of the eight volumes $D0$ through $D7$. The left and center columns show the original and SRCMap-modified load (IOPS) for each volume. The modified load were consolidated on disks $D2$ and $D3$ by SRCMap.

Note that disks $D6$ and $D7$ are continuously in standby mode, $D3$ is continuously in active mode throughout the 24 hour duration while the remaining disks switched states more than once. Of these, $D0$, $D1$ and $D5$ were maintained in standby mode by SRCMap, but were spun up one or more times due to read misses to their replica volumes, while $D2$ was made active by SRCMap for two of the consolidation intervals only.

We note that the number of spin-up cycles did not exceed 6 for any physical volume during the 24 hour period, thus not sacrificing reliability. Due to the reliability-aware design of SRCMap, volumes marked as active consume power even when there is idleness over shorter, sub-interval durations. For the right column, power consumption for each disk in either active mode or spun down is shown with spikes representing spin-ups due to read misses in the volume's active replica. Further, even if the working set changes drastically during an interval, it only leads to a single spin up that services a large number of misses. For example, $D1$ served approximately $5 * 10^4$ misses in the single spin-up it had to incur (Figure omitted due to lack of space). We also note that summing up power consumption of individual volumes cannot be used to compute total power as per Table 3.3(b).

Sensitivity with Space Overhead We evaluated the sensitivity of SRCMap energy savings with the amount of over-provisioned space to store volume working sets. Figure 3.12 depicts the average power consumption of the entire storage system (i.e., all eight volumes) across a 24 hour interval as the amount of over-provisioned space is varied as a percentage of the total storage space for the load level $L0$. We observe that SRCMap is able to deliver most of its energy savings with 10% space over-provisioning and all savings with 20%. Hence, we conclude that SRCMap can deliver power savings with minimal replica space.

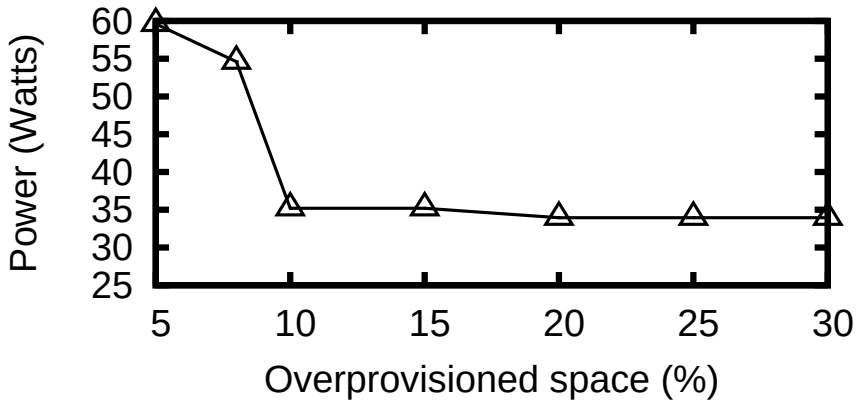


Figure 3.12: Sensitivity to over-provisioned space.

Energy Proportionality Our next experiment evaluates the degree of energy proportionality to the total load on the storage system delivered by SRCMap. For this experiment, we examined the power consumption within each 2-hour consolidation interval across the 24-hour duration for each of the five load estimation levels L0 through L4, giving us 60 data points. Further, we created a few higher load levels below $L0$ to study energy proportionality at high load as well. Each data point is characterized by an average power consumption value and a *load factor* value which is the observed average IOPS load as a percentage of the estimated IOPS capacity (based on the load estimation level) across all the volumes. Figure 3.13 presents the power consumption at each load factor. Even though the load factor is a continuous variable, power consumption levels in SRCMap are discrete. One may note that SRCMap can only vary one volume at a time and hence the different power-performance levels in SRCMap differ by one physical volume. We do observe that SRCMap is able to achieve close to N-level proportionality for a system with N-volumes, demonstrating a step-wise linear increase in power levels with increasing load.

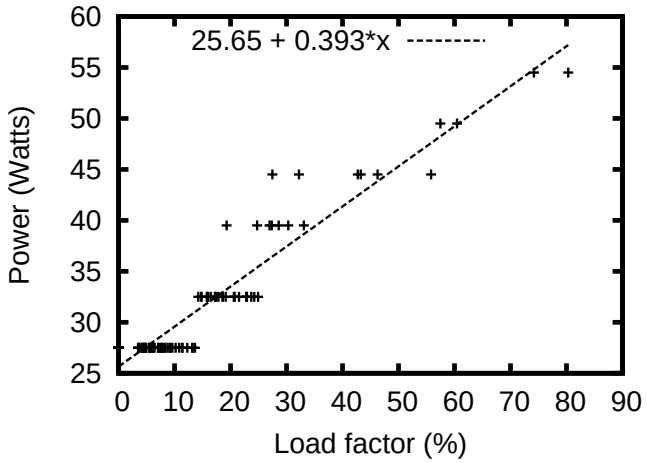


Figure 3.13: Energy proportionality with load.

3.7.3 Resource overhead of SRCMap

The primary resource overhead in SRCMap is the memory used by the *Replica Metadata (map)* of the *Replica manager*. This memory overhead depends on the size of the replica space maintained on each volume for storing both working-sets and off-loaded writes. We maintain a per-block map entry, which consists of 5 bytes to point to the current active replica. 4 additional bytes keep what replicas contain the last data version and 4 more bytes are used to handle the I/Os absorbed in the replica-space write buffer, making a total of 13 bytes for each entry in the map. If N is the number of volumes of size S with R% space to store replicas, then the worst-case memory consumption is approximately equal to the *map* size, expressed as $\frac{N \times S \times R \times 13}{2^{12}}$. For a storage virtualization manager that manages 10 volumes of total size 10TB, each with a replica space allocation of 100GB (10% over-provisioning), the memory overhead is only 3.2GB, easily affordable for a high-end storage virtualization manager.

3.8 Summary

This chapter presented the design and evaluation of SRCMap, a storage virtualization solution for energy-proportional storage in multi-disk systems. SRCMap establishes the feasibility of an energy proportional storage system with fully flexible dynamic storage consolidation. SRCMap is able to meet all the desired goals of fine-grained energy proportionality, low space overhead, reliability, workload shift adaptation, and heterogeneity support.

SRCMap makes energy-proportionality attainable in multi-disk system by providing one energy-level per disk. The benefits are directly proportional to the number of disks available in the system. Consequently, SRCMap is unable to offer benefits in the case of single-disk systems. Saving energy in single-disk systems can still have a high impact due to the widespread use of portable devices. In the next chapter we explore a new technique for reducing energy consumption in such devices.

3.9 Credits

SRCMap was first published in the proceedings of the *USENIX Conference on File and Storage Technologies* in February 2010 [VKUR10] and was presented by Luis Useche. Akshat Verma contributed the initial formulation of the SRCMap optimization framework and algorithms for the initial replica placement and active volume identification. All the authors helped to refine the initial design of the system. Ricardo Koller implemented a simulator and used it to evaluate the optimization framework and algorithms. Luis Useche designed and implemented a prototype of SRCMap and used it to evaluate the system on real hardware.

CHAPTER 4

ENERGY-EFFICIENT STORAGE USING FLASH

Chapter 3 demonstrated the feasibility of energy proportional multi-disk systems. However, because disks operate on only one energy level, such energy proportionality cannot be achieved in single-disk systems. In this chapter we present the design, implementation, and evaluation of a new technique to save energy when only one disk is available. This solution uses a persistent external caching device that absorbs I/O activity, providing the disk with longer periods of idle time where the disk can be spun down. The benefits are particularly relevant for the widely available portable systems where energy is usually scarce.

4.1 Overview

The need for energy-efficient storage systems for both personal computing and data center environments has been well established in the research literature. The key argument is that the disk drive, the sole mechanical device in modern computers, is also one of its most energy consuming [BH09]. The varied proposals for addressing this problem include adaptive disk spin down policies [HLSS00, DKB95], exploiting multi-speed drives [GSKF03], using data migration across drives [PB04], and energy-aware prefetching and caching techniques [PS04, Sam04].

A different approach, complementary to most of the above techniques, is *external caching*[1] on a non-volatile storage device, which we shall henceforth refer to as *external caching device* (ECD). An ECD can be any non-volatile device that consumes less energy than a disk drive, such as flash. Recent technological advancements, adoption trends, and economy-of-scale benefits have brought the non-volatile flash-based storage devices into the mainstream. Recent work on external caching have

[1] We term this as "external caching" to primarily differentiate it from in-memory caching.

presented the merits of such systems. While these studies serve to make the case for further research in external caching systems, they still leave several key questions unanswered. First, these studies do not evaluate the energy consumption of the system as a whole, but only focus on the reduction in disk energy consumption. It is important to refine this evaluation criteria since the ECD subsystem itself can consume a considerable amount of energy. Second, existing studies do not evaluate an important artifact of external caching, which is the impact on application performance. Flash-based devices handle random reads much better than disk drives, but perform slightly worse than disk drives for sequential accesses and substantially worse for random writes [EM05]. Third, the existing approaches base their evaluation of external caching on simulation models [BBL06, CJZ06, MDK94]. While simulation-based evaluation may be well-suited for an approximate evaluation of a system, they also sidestep key design and implementation complexities as well as preclude evaluating the overhead contributed by the system itself.

In this chapter we describe the design and implementation of EXCES, an external caching system for energy savings, that comprehensively addresses the above questions and advances the state of our understanding of external caching systems. EXCES operates by utilizing an ECD for prefetching, caching, and buffering of disk data to enable the disk to be spun-down for large periods of time and save energy. EXCES is an online system — it adapts to the changing workload by identifying popular data continuously, reconfiguring the contents of the ECD (as and when appropriate) to maximize ECD hits on both read and write operations. To prefetch popular data which are not present in the ECD, EXCES opportunistically reconfigures the ECD contents, when the disk is woken up on an ECD *read miss*. EXCES always redirects writes to the ECD, regardless of whether the written blocks were prefetched/cached in the ECD; this is particularly important since most systems per-

form background write IO operations, even when idle [CJZ06, PADAD05, Sam04]. All of the above optimizations minimize disk accesses and prolong disk idle periods, consequently conserving energy.

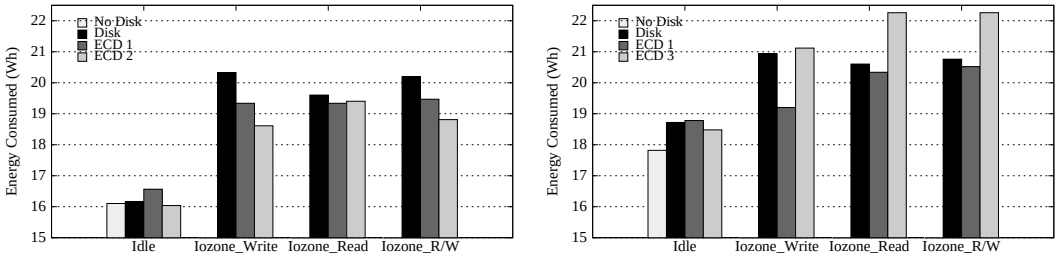
We implement EXCES as a Linux kernel module to demonstrate the suitability of external caching in production systems. EXCES operates between the file system and I/O scheduler layers in the storage stack, making it independent of the filesystem and availing kernel I/O scheduling automatically. EXCES provides strong block-layer data consistency for all blocks managed by upper layers, by maintaining a persistent page-level *indirection map*. It successfully addresses the challenges of page indirection, including partial/multiple block reads and writes, optimally flushing dirty pages to the disk drive during reconfiguration, correctly handling foreground accesses to pages that are undergoing reconfiguration, and ensuring "up-to-dateness" of the indirection map under all these conditions.

We evaluated EXCES for different workloads including both micro-benchmarks and laptop-specific benchmarks. In most cases, EXCES was able to save a reasonable amount of energy (∼2-14%). However, we found that using a flash-based ECD can substantially degrade I/O performance and careful consideration is needed before deploying external caching, especially in performance-centric data center environments. Finally, we measured the resource overheads incurred due to EXCES and found these well within acceptable limits.

In Section 4.2, we profile the energy consumption of disk drives, ECDs and ECD interfaces, on two different systems. Section 4.3 presents the architecture of EXCES. Section 4.4 presents the detailed design and Section 4.5 overviews our Linux kernel implementation of EXCES. In Section 4.6, we conduct an extensive evaluation of EXCES.

Configuration	Disk State	Iozone Data	ECD Specification
No Disk	Standby	N/A	N/A
Disk	Active	On disk	N/A
ECD 1	Standby	On ECD	SanDisk Cruzer Micro USB
ECD 2	Standby	On ECD	SanDisk Ultra CF Type II
ECD 3	Standby	On ECD	SanDisk Ultra CF Type II

Table 4.1: Various laptop configurations used in profiling experiments.



(a) The *shiriu* system (b) The *beer* system

Figure 4.1: Energy consumption profiles of various ECD types and interfaces.

4.2 Profiling Energy Consumption

To understand the energy consumption characteristics of ECD relative to disk drives, we experimented with two different NAND-flash ECDs and three different ECD interfaces on two laptop systems. Table 4.1 shows the different configurations of the devices used in the profiling experiments.[2] All ECD devices were 2GB in size. We measured the overall system energy consumption for four states: when the system was idle with each device merely being active, and with the Iozone [NC], an I/O intensive benchmark, generating a read intensive, write intensive, and read-write workload.

Figure 4.1 depicts the individual energy consumption profiles for each storage device on two different laptops: `shiriu` and `beer`. A detailed setup of each machine

[2]We also tried using an SD NAND flash device. Unfortunately its Linux driver is still under development and performs poorly for writes (<4 KB/s); consequently, we discontinued experiments with that device.

is given in Section 4.6 (Please see Table 4.2). During each experiment exactly one device is turned on. These experiments were conducted using a Knoppix Live CD to enable complete shutdown of the disk when not being tested.

It can be observed that each machine has a distinct behavior. On the `shiriu` system, the USB subsystem consumes substantially more energy than the disk subsystem when the system is idle; we believe this is partly due to an unoptimized driver for the Linux kernel [Bro04]. However, both types of flash memory consume less energy than the disk in all the Iozone benchmarks. On the `beer` system, the findings were somewhat surprising. Although the exact same flash device was used in the ECD 2 and ECD 3 configurations, the PC Card interface in the ECD 3 configuration negatively impacted energy consumption in all the Iozone benchmarks. While we do not know the exact cause, we postulate this could be due to an unoptimized device driver.

More importantly, for both systems, even in configurations when the disk is powered down completely, we observe that the energy savings are bound within 10% for an I/O intensive benchmark. Further, when the system is idle, the ECD subsystems consumes as much energy as the disk drive. While the laptop workload would be somewhere in between idle and I/O intensive, these findings nevertheless call to question the effectiveness of *external caching* systems in saving energy. Our goal in this study is to address this question comprehensively.

4.3 System Architecture

Figure 4.2 presents the architecture of EXCES in relation to the storage stack within the operating system. Positioning EXCES at the block layer is important for several reasons. First, this allows EXCES coarse-grained monitoring and control over system devices, at the *block device* abstraction. Additionally, the relatively simple block

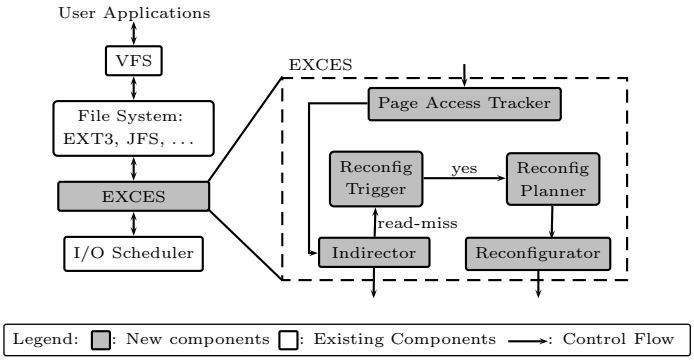


Figure 4.2: EXCES system architecture.

layer interface allows easy I/O interception and indirection, and also allows EXCES to be designed as a dynamically loadable kernel module, with no modifications to the kernel. Second, by operating at the block layer, EXCES becomes independent of the file system, and can thereby work seamlessly with any file system type, and support multiple active file systems and mount-points simultaneously. Third, internal I/Os generated by EXCES itself leverage the I/O scheduler, automatically addressing the complexities of block request *merging* and *reordering*.

EXCES consists of five major components as shown in Figure 4.2. Every block I/O request issued by the upper layer to the disk drive is intercepted by EXCES. The *page access tracker* receives each request and maintains updated popularity information at a 4KB page granularity. Control subsequently passes to the *indirector* component which redirects the I/O request to the ECD as necessary. Read requests to ECD cached blocks and all write requests are indirected to the ECD. A *read-miss* occurs for blocks not present on the ECD and the read request is then indirected to the disk drive. The *reconfiguration trigger* module is invoked which decides if the state of the system necessitates a reconfiguration operation. If a reconfiguration is required, the *reconfiguration planner* component uses the page rank information maintained by the page access tracker to generate a new reconfiguration plan which

contains the popular data based on recent activity. The *reconfigurator* uses this plan and performs the corresponding operations to achieve the desired state of the ECD. EXCES continuously iterates through this process until the EXCES module is unloaded from the kernel.

4.4 System Design

In designing EXCES, we used the following behavioral goals as guidelines: *(i)* increase disk inactivity periods through data prefetching, caching, and write buffering on the ECD, *(ii)* make more effective use of the ECD by continuously adapting to workload changes, *(iii)* ensure block-level data consistency under all system states, and *(iv)* minimize the system overhead introduced due to EXCES itself. In the rest of this section, we describe how the various architectural components of EXCES work towards realizing these design goals.

4.4.1 Page Access Tracker

The *page access tracker* continuously tracks the popularity of the pages accessed by applications. We track popularity at the *page granularity* (instead of block granularity, the unit of disk data access) to utilize the fact that file systems access the disk at the larger page granularity for most operations. This reduces the amount of EXCES metadata by a factor of 8X.

Page popularity is tracked by associating with each page a *page rank*. In our initial study we found that while accounting for recency of access was important for a high ECD hit ratio, there were certain pages that were accessed periodically. LRU-type main memory caching algorithms, tuned to minimize the total *number* of disk accesses, typically end up evicting such pages prematurely for large working-set sizes. In the case of external caching systems, if this page is not present in the

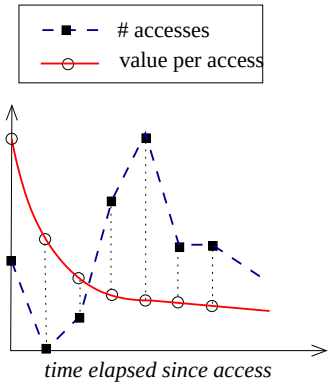


Figure 4.3: Page rank decay function

ECD the disk will need to be woken up to service it periodically, thereby leaving little opportunity for energy savings. Consequently, the page ranking mechanism in EXCES provides importance to both recency and frequency of accesses to determine the rank of a page.

For each page P, the page ranking mechanism splits time into discrete quanta (t_i) and records the number of accesses to the page within each quantum (a_i^P). When the rank for a page must be updated, the page ranking mechanism weights page accesses over the time-line using an exponential decay function (f) as shown in Figure 4.3. The rank of a page P is obtained as $rank(P) = \sum a_i^P \cdot f(t_i)$. The page ranking mechanism thus awards a higher value for recent accesses, but also takes into account frequency of accesses, by retaining a non-trivial value for accesses in the past.

4.4.2 Indirection

The *indirector* is a central component of EXCES. Similar to the page access tracker, it gets activated upon each I/O request to appropriately redirect it to the ECD if required.

```
Require: Page Request: req, Indirection Map: map.
 1: if req does not contain an entry in map then
 2:     if req is write then
 3:         if disk state is STANDBY then
 4:             find free (alternatively clean) page in ECD
 5:             if page in ECD is found then
 6:                 add new entry in map (mark dirty)
 7:                 change the req location as per map entry
 8: else
 9:     change the req location as per map entry
10: send request req
```

Algorithm 2: Indirection Algorithm

The indirector maintains an *indirection map* data structure to keep track of disk pages that have been prefetched, cached, or buffered in the ECD. Each entry in the indirection map includes the *disk page* mapped, the corresponding *ECD page* where it is mapped to, and whether the copy in the ECD is *dirty* or not. The data structure is implemented so that we can find a specific entry, either given the page information on the ECD or the page on disk. EXCES uses native kernel data structures that allow constant time operations for the above.

For each I/O request, the indirector component first checks to see if it is larger than a page. If so, it splits it into multiple requests, one for each page. Each page request is handled based on four factors: *(i)* type of operation (read or write), *(ii)* the disk power state, *(iii)* indirection map entry, and *(iv)* presence of free/clean page in the ECD. Algorithm 2 shows the algorithm followed by the indirector for each page request. The algorithm attempts to keep the disk in idle state as long as possible, to maximize energy savings. This is feasible in two cases - if there is a free or clean page in the ECD (line 5) to absorb a page write request, or if the page is already mapped (line 9).

In the rest of the cases, the disk is either active or would have to be spun up owing to an ECD miss. In each such case, the ECD miss counter is incremented;

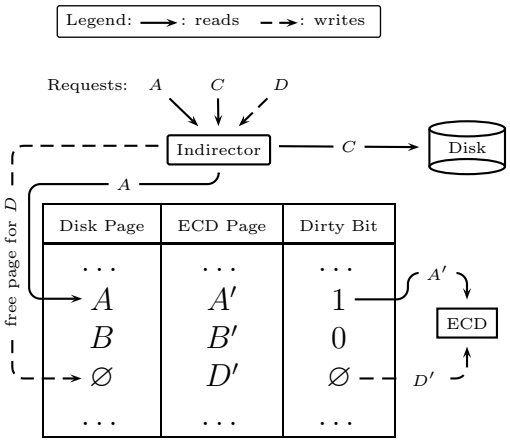


Figure 4.4: Indirection example

this counter is used by the *reconfiguration trigger* component of EXCES (described shortly). In the example of Figure 4.4, there are three page requests: A, C and D. In the case of A, there is an entry in the indirection map; consequently, it gets indirected to the corresponding page in the ECD. In the case of C, the page does not have an entry in the indirection map; the indirector lets the request continue to the disk. Finally, the write request D is handled differently than the above. There is no map entry for D. However, having found a free page in the ECD, the indirector creates a new map entry and redirects the request to the ECD, thereby avoiding spinning up the disk.

4.4.3 Reconfiguration Trigger

Upon each ECD miss, the indirector invokes the *reconfiguration trigger*, which determines if a reconfiguration of the ECD contents would be appropriate at the current time. If yes, it invokes the *reconfiguration planner* component (described next); otherwise, it does nothing.

The appropriateness of a reconfiguration operation depends on three necessary conditions: *(i)* the target state of the ECD contents is different than the current

one; *(ii)* the current ECD miss rate (per unit of time) has exceeded a threshold, and *(iii)* a threshold amount of time has elapsed since the previous reconfiguration operation. If the above hold true, the reconfiguration trigger concludes that the current state of ECD contents is not favorable to energy saving, and consequently must be reconfigured to reflect recent changes in the workload.

4.4.4 Reconfiguration Planner

The *reconfiguration planner* creates a list of operations, which constitute the *reconfiguration plan*, to be performed during the next reconfiguration operation. To be able to create such a list whenever invoked, it continuously maintains a *top-k matrix* data structure, that holds the "top k" ranked pages. Choosing k as the size of the ECD in pages, this matrix can then be used to identify the target contents for the ECD that the reconfiguration operation must achieve.

The top-k matrix continuously incorporates the page rank updates provided by the page access tracker. The threshold for being inserted into the top-k matrix is set by its lowest ranked page. (We present and analyze this data structure in detail in Section 4.5).

The reconfiguration plan is constructed in two parts. The first are the "outgoing" pages which must be flushed to the disk; these are no longer popular enough to be in the ECD and are dirty. The second are the "incoming" pages which now have a sufficiently high rank to be in the ECD but are not currently in it. These constitute the pages to be *prefetched* to ensure a high ECD hit ratio in the future.

Construction of the reconfiguration plan occurs upon invocation by the reconfiguration trigger. The outgoing and incoming lists are then created based on the top-k matrix contents and the indirection map. The reconfiguration planner walks through each page of the ECD, creating an entry in the outgoing list for each page

that is no longer in the top-k matrix. Next, it walks through each entry in the top-k matrix, creating an entry in the incoming list for each page that is currently not in the ECD. Once these two stages are completed, the new reconfiguration plan is obtained.

4.4.5 Reconfigurator

The *reconfigurator* component of EXCES performs the actual data movement between the disk and ECD. Broadly, the goal of each reconfiguration operation is to reorganize the ECD contents based on changes in the application I/O workload, so that disk idle periods are prolonged. This is done simply by following the reconfiguration plan as created by the planner component.

Require: Phase: *phase*, Origin: *orig*, Destination: *dest*, Indirection Map: *map* Table: *map*
1: **if** $phase = DISK\_TO\_ED$ **then**
2: add a new entry $[orig, dest]$ in the *map*
3: mark the new entry as "clean"
4: read from *orig*
5: write to *dest*
6: **if** $phase = ED\_TO\_DISK$ **then**
7: delete the entry for *dest* from the *map*

Algorithm 3: Algorithm used for a single operation during reconfiguration.

The reconfiguration operation is managed in two distinct "phases": *ECD to Disk* and *Disk to ECD*. These two phases are treated differently, and are detailed in Algorithm 3. The first phase, *ECD to Disk*, addresses operations in the *outgoing* list of the reconfiguration plan. For each entry in the list, the data movement operation is *followed by* deleting the corresponding entry in the indirection map. The second *Disk to ECD* phase, handles the incoming list in a similar way, except that a new entry is added to the indirection map, *prior to* the actual data movement.

Indirection during reconfiguration Indirecting I/O requests issued by applications during the reconfiguration operation must be carefully handled due to implicit race conditions. A race condition arises if an application accesses a page currently being reconfigured. While it is perhaps simpler to postpone servicing the application I/O request until the reconfiguration operation for the page is completed (to ensure data consistence), this delay can be avoided. We designed separate policies to handle *read* and *write* operations issued by the application. If the application issued a *read* request, the indirector issues the read to the *origin* page location so it provides the most up-to-date data. For a *write* request by the application, the request is issued to the *dest* location and the reconfiguration for the page is discontinued. These policies help to alleviate the overhead the reconfiguration causes to the user level applications by minimizing I/O wait time for foreground I/O operations.

4.4.6 Other Design Issues

Disk spin-down policy Researchers have proposed two classes of policies: *dynamic* and *static* [HLSS00, DKB95]. In dynamic policies, the system dynamically varies the time the disk needs to stay idle before being put on standby. In EXCES we chose to use a static policy with a fixed timeout for spin-down. In the evaluation section, we experiment with various values for this timeout parameter.

Data Consistency Data consistency is always an important issue whenever multiple copies of the same information exist. In EXCES, data is replicated in the ECD. We need to ensure that the system reads up-to-date versions of data after rebooting the machine as well as in case of system crash or sudden power failure. We reserve the first portion of the ECD to maintain a persistent copy of the indirection map.

```
typedef struct {
  unsigned int rank;
  unsigned int tmp_rank;
  unsigned short last_acc[H_SIZE];
  unsigned int disk_lbn;
} page_ranker_t
```

Figure 4.5: The `page_ranker` structure

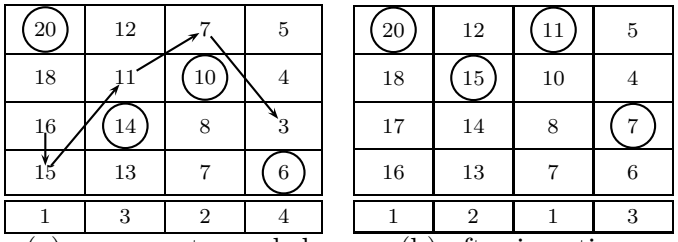
This persistent copy of the indirection map is updated each time the map is changed and gets invalidated if the EXCES kernel module is unloaded cleanly. In case of a power failure or system crash, all entries contained in the persistent indirection map are assumed to be dirty.

4.5 System Implementation

We implemented EXCES as a Linux kernel module that can be dynamically inserted and removed without any changes to the kernel source. Since the block layer interface of the Linux kernel is very stable, EXCES can run "out of the box" on the latest 2.6 series kernels. The current implementation of EXCES utilizes native kernel data structures such as *radix trees* and *red-black trees* which are very likely to be retained in the future kernel versions. In this section, we elaborate on key aspects of the EXCES system implementation that are novel and those which were particularly challenging to "get right".

4.5.1 Maintaining the Top-k Ranked Pages

The EXCES page ranking mechanism (described in Section 4.4.1) considers both recency and frequency of page accesses. Figure 4.5 shows the `page_ranker_t` structure that is used to encapsulate the rank of a page. This structure allows us to efficiently capture the history of page rank values updated due to accesses over



(a) movements needed (b) after insertion

Figure 4.6: The Top-k matrix. Figure 4.6(a) shows the matrix before the insertion and indicates the necessary movements and the resulting matrix after inserting the entry 17 is shown in Figure 4.6(b).

time. `disk_lbn` stores the starting on-disk *logical block number* of the page and `last_acc` array contains the timestamps of the last `H_SIZE` accesses (default is 4). Each time the `last_acc` array is filled up, it is passed to the ranking decay function (Figure 4.3); the resulting values are stored in `tmp_rank` using a compact representation and the `last_acc` array is reset. Before overwriting the `tmp_rank`, its previous value is decomposed and added to the historical rank of the page contained in `rank`. The *actual* rank of a page at any time is given by the sum of decomposed `tmp_rank` and `rank` values.

To be able to access the top-k ranked pages (whenever required by the reconfiguration planner), we implemented the *top-k matrix*, a novel matrix data structure of dimensions $(\sqrt{k}+1) \times (\sqrt{k})$, which stores the top-$k$ ranked pages. Since k can be large (as much as 10^8 for gigabyte-sized ECDs), operations on the top-k matrix must be highly efficient. While regular sorted matrices are good for lookups ($O(log(\sqrt{k}))$ using binary search in both columns and rows), insertions are expensive at $O(k)$ since all the lower (or upper) values must be shifted. To reduce the insertion cost, we use an extra row to store an offset that indicates where the maximum value of the column is located; all the elements of that column are also sorted according to that offset. (Please see Figure 4.6(b) for an example.) By maintaining this extra in-

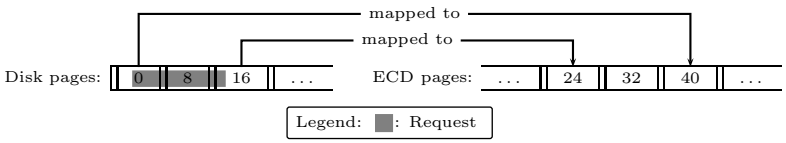


Figure 4.7: Alignment problem example

formation we retain $O(log(\sqrt{k}))$ lookups and can also perform insertions in $O(\sqrt{k})$. This is because, in the worst case, we need to shift elements in exactly half a column and transfer its minimum to the next column where it becomes the maximum, and so on, until we reach the last column. A detailed example of the worst-case insertion process into the top-k matrix is presented in Figure 4.6.

Detecting if a page belongs in the top-k highest ranked pages is as easy as checking if its rank is greater than the minimum rank in the top=k matrix, in which case the page must be inserted, and marked for inclusion in the next round of reconfiguration.

4.5.2 Indirector Implementation Issues

As mentioned earlier, in EXCES we chose to maintain metadata about data popularity and data replication at the granularity of a *page*. While this optimization allows us to drastically cut down on metadata memory requirement (by 8X), it complicates the implementation of the indirector component. Since I/Os may be issued at the block granularity, the indirector component must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. In EXCES, we address this issue via I/O request splitting and page-wise indirection.

Figure 4.7 shows an example of the alignment problem that the indirector must handle. Notice that two pages on the disk mapped to the ECD. The first page on the disk that starts at block 0, is mapped to the fifth page on the ECD that starts

from block 40. Also, the third page in the disk (starting at block 16), is mapped to the fourth page of the ECD, starting at block 24 of the ECD. The second page in the disk is not mapped to the ECD at all.

Consider an application I/O request as represented by the shaded region. This request covers a part of the first, the entire second page, and a part of the third page on disk. The indirection operation is complicated because the I/O request is not page-aligned. The indirector must individually redirect each part of the request to their appropriate locations. The above can occur with both read and write I/O requests.

In EXCES, to address I/O splitting, we create new requests using the Linux kernel *block I/O* structure called `bio`, one per page. All attributes of the `bio` structure are automatically populated based on lookups to the *indirection map*, including the sector, offset, and length within the page that will be filled/emptied depending of the operation. After the splitting and issuing each "sub-I/O", the indirector waits for all sub-I/Os to complete before notifying the requester about the completion of the original I/O operation.

There is a special case while handling write requests that are not already mapped to ECD and that are not page-aligned. If EXCES buffers such writes in the ECD (as it does with other page-aligned writes), there will be inconsistency since a portion of the page will hold invalid data. For this special case, we let the request continue to disk.

4.5.3 Modularization and Consistency

EXCES utilizes the design of the block layer inside the Linux kernel to enable its operation as a dynamically loadable kernel module. Specifically, each instantiated block device registers a kernel function called `make_request` that is used to handle

the requests to the device. EXCES is dynamically included in the I/O stack by substituting the `make_request` function of the disk device targeted for energy savings. This allows us to easily and directly modify any I/O requests before they are forwarded to the disk.

While module insertion is simple enough, module removal/unload must bear the additional responsibility of ensuring data consistency. Upon removal, EXCES must flush on-ECD dirty blocks to their original positions on disk. In EXCES, the I/O operations required to flush dirty pages upon module unload are handled using the reconfigurator through the *ECD to Disk* phase. In addition, EXCES must address race conditions caused when an application issues an I/O request to a page that is being flushed to disk at that exact instant. To handle such races, EXCES stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are committed to the disk. Since we expect module unload to be a rare event and the probability that a request for a page at the exact time it is being flushed to be low, the average response time for application I/O remains virtually not affected.

4.6 Evaluation

In our evaluation of the EXCES system, we answer the following questions in sequence: *(i)* What is an appropriate spin-down timeout for EXCES? *(ii)* How much energy does EXCES save? *(iii)* What is the impact on application I/O performance when EXCES is used? and *(iv)* what is the overhead of the EXCES system in terms of memory and computation?

To assess the above, we conducted experiments on two laptops, `shiriu` and `beer` (Table 4.2), both running Linux kernel 2.6.20. The experiments utilized the ECDs described in Table 4.1. The hard drives on both laptops were running the Linux ext3 file system and the ECDs used ext2.

To quantify the system's energy dissipation we used the battery information provided by ACPI, and took readings of the energy level at 10 seconds intervals, the default ACPI update interval. The display brightness was reduced to its minimum visible level in all experiments.

For comparison purposes, in each experiment we set up various configurations including a default system with no optimizations, a system configured with the *laptop-mode* energy saving solution [Sam04], a system configured with EXCES, and a system configured with both laptop-mode and EXCES. Laptop-mode is a setting for the Linux kernel that forces much larger read-aheads (default 4MB) and holds off writes to the disk by buffering them in memory for a much longer time period. In all experiments, EXCES was configured to use an ECD miss rate threshold of 1000 misses-per-minute to trigger reconfiguration and a minimum duration of one minute between two reconfiguration operations.[3]

We used the BLTK (Linux Battery Life Tool Kit) [BKL+07] as our primary benchmark for system evaluation. This benchmark focuses specifically on laptop-specific workloads, targeted for evaluating battery life of laptop systems in realistic usage scenarios. Specifically, we use the BLTK Office and the BLTK Developer benchmarks in our experiments. Additionally, we use the Postmark [Kat97] file system benchmark, designed to simulate small file workloads, typical of an email server. While we do not suggest the use of EXCES on the server-side (as yet), this benchmark allows us to evaluate the impact of I/O intensive workloads on external caching systems.

[3]While we used these static values (based on preliminary experimentation) for simplicity, subsequent versions of EXCES will be able to dynamically adapt these thresholds based on application workload.

Name	Model	CPU	RAM	HD Specifications
shiriu	Dell E1505	Intel Core 2 @ 1.83 GHz	1 GB	Toshiba MK1234GSX (5400 RPM, 120 GB)
beer	Dell 600m	Intel Pentium M @ 1.6 GHz	512 MB	Western Digital WD400VE (5400 RPM, 40 GB)

Table 4.2: Specifications of the machines used in the experiments

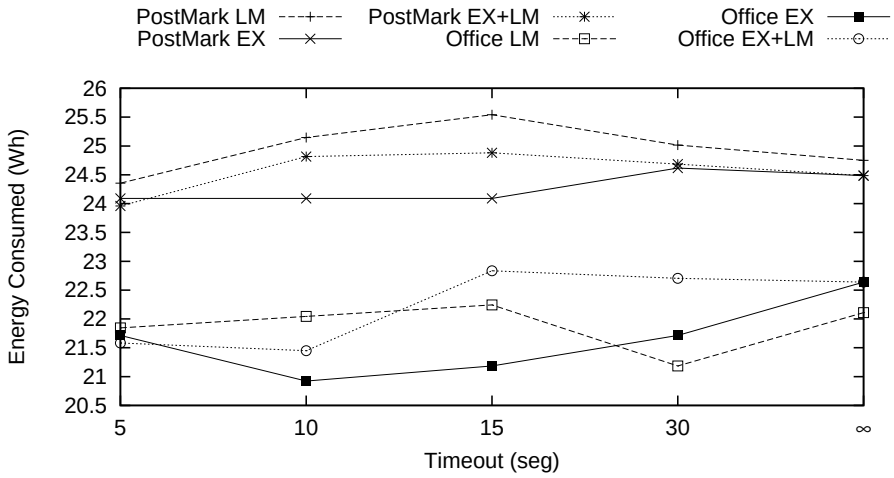


Figure 4.8: Effect of the disk spin-down timeout value on energy savings

4.6.1 Choosing the Disk Spin-down Timeout

While researchers have suggested the benefits of using adaptive disk spin-down timeouts [HLSS00, DKB95], the current version of EXCES uses a static disk spin-down timeout value. We used two benchmark workloads to determine the effect of the spin-down timeout on energy savings, using the `shiriu` system. We compared the system when configured with EXCES, laptop-mode [Sam04], and a combination of EXCES with laptop-mode. We used the *ECD 2* configuration from Table 4.1 for this experiment. The BLTK Office benchmark, which automates the activities of opening and editing OpenOffice.org documents, spreadsheets, and drawings, was our first workload. The second workload used was the PostMark benchmark.

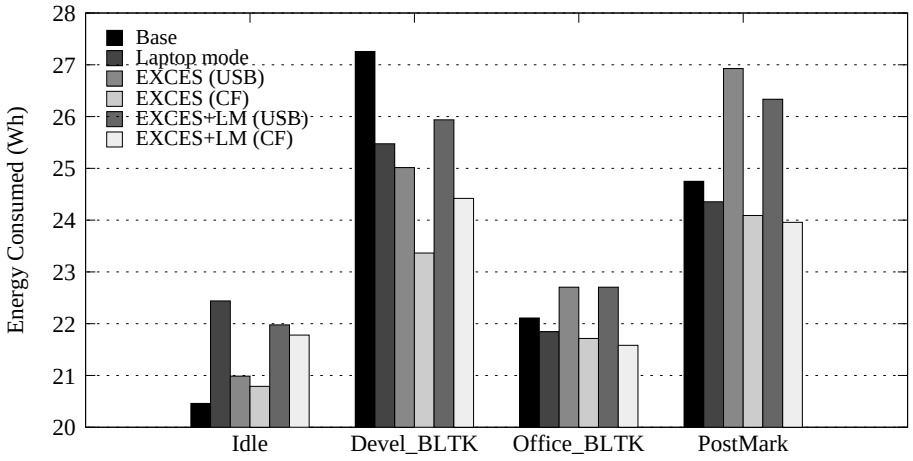


Figure 4.9: Energy consumption with different workloads

Figure 4.8 shows the results using timeout intervals of 5, 10, 15, 30 seconds and no timeout (∞ seconds). We used `hdparm` to set the timeout intervals in the disk's firmware, restricted to a minimum value of 5 seconds. A general trend observed when using EXCES (with and without laptop-mode) is that smaller timeout values allowed for greater energy savings, except for the BLTK Office workload which reaches its optimum at 10 seconds. All subsequent experiments use a 5 second disk spin-down timeout.

4.6.2 Energy Savings

To evaluate EXCES savings, we measure the energy consumption with six different system configurations: the system "as is" with no energy saving solution (Base), system running Laptop mode, EXCES using a USB and CF as ECD respectively, and EXCES with Laptop mode activated, using a USB and CF as ECD respectively. These experiments were conducted on the `shiriu` system, using configurations *ECD 1* for USB and *ECD 2* for CF (per Table 4.1).

We evaluated four different workloads: *(i)* an idle system, *(ii)* the BLTK Developer benchmark, *(iii)* the BLTK Office benchmark, and *(iv)* the Postmark benchmark. Figure 4.9 shows that on an idle system, all energy saving systems consume more energy that the Base configuration. The laptop-mode configuration uses additional energy because of its aggressive prefetching mechanism, which ends up waking the disk for unnecessary data fetch operations. A similar behavior is observed when using EXCES in combination with laptop-mode. When EXCES is used by itself, since the disk is mostly spun-down anyway, any small disk energy savings is negated by the extra energy consumed due to the ECD device itself.

The BLTK Developer benchmark performs a moderate amount of I/O. Its behavior mimics the operations of a developer who creates new files, edits files using the `vi` editor, and compiles the Linux kernel source tree, all of these interspersed with appropriate "human like" idle periods. We notice that moderate energy savings can be obtained for all the energy-optimized solutions. The configuration with EXCES alone provides the most energy savings ($\sim$14% with CF and $\sim$8% with USB). The configurations that use laptop-mode deliver relatively lesser energy savings; we attribute this to a fraction of the prefetching operations turning out to be ineffective.

For the BLTK Office benchmark, we note that there is no substantial energy saving in any of the configurations, and energy consumption is somewhat increased when using a USB device as a ECD. We believe that this is due to the behavior of the benchmark which opens large executables from the OpenOffice suite, typically stored sequentially on the disk drive. These are subsequently cached in main memory, resulting in very few I/O operations after the executables have been loaded, reducing the opportunity for energy saving.

Finally, energy consumption for the Postmark benchmark follows a similar trend to the BLTK Office. However, in this case, we believe the reasoning is different.

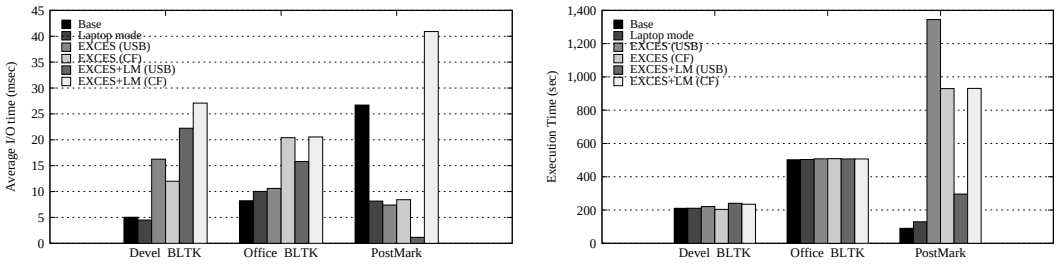
Postmark is an I/O intensive workload, with a large fraction of sequential write operations. These sequentially written blocks to disk get absorbed as random writes in the ECD, owing to the current implementation of write buffering which does not attempt to sequentialize buffered writes. Random writes on flash-based storage are the least efficient, both in performance and energy consumption [EM05]. We believe that a better implementation of write buffering in EXCES which improves the sequentiality of buffered writes, can result in better energy savings for a write intensive workload.

It is interesting to note that in almost all the cases using the USB as ECD makes the energy saving system to consume more energy than the base case. On the other hand, using the CF leads to a better results for EXCES. Further, our findings point to a range of ∼2-14% for the cases when EXCES was indeed able to reduce energy consumption, in somewhat of a contrast to earlier results from simulation studies [BBL06, CJZ06, MDK94] that predicted energy savings of ∼20-46%[4]. This difference is primarily because the energy-consumption of the ECD was considered negligible and ignored in those studies.

4.6.3 Performance Impact of External Caching

While ECDs offer better performance than disk drives for random reads, they performs worse for other workloads. To evaluate performance, we focus on two metrics: *(i)* the average I/O (completion) time, and *(ii)* overall benchmark execution time. These provide complementary information and allow us insight into I/O performance. The average I/O time was obtained by using the Linux kernel tool *blktrace* [ABO07]. The benchmark execution time was measured using the Bash *time* command. Each benchmark was run several times and the results averaged.

[4]extrapolated to address whole system energy consumption



(a) Average I/O time (b) Benchmark Execution time

Figure 4.10: Performance impact of EXCES with various workloads

Figure 4.10 shows the results of these experiments. In both the BLTK benchmarks, the average I/O time increases substantially for the ECD based solutions, due to a large fraction of the I/O workload being sequential, allowing the disk drive to perform better. However, the increase in the overall benchmark execution time is negligible, due to substantial idle periods between I/O operations. This indicates that the impact to user perceived performance is minimal, thereby making the case for using external caching with these laptop-oriented workloads. On the other hand, an interesting anomaly is observed for the write intensive server-oriented Postmark benchmark. While the average I/O times with most of the ECD based solutions are lower, the total execution times are higher. We believe that this counter-intuitive result is because of writes being reported as completed by the ECD when they are written to the cache on the ECD but before they are actually committed to the flash medium. This reporting mechanism gives the false impression of fast individual write operations, when in reality the overall write I/O performance is severely throttled as the ECD commits these writes to the persistent flash with high latency.

4.6.4 Resource Overhead

In EXCES, we paid special attention to how we used the scarce kernel-space memory. The memory usage for each of the EXCES's data structures is presented in the Table

Pages/GB	Dirty-bit array	Indirection map	Phase table	Top-k matrix	Page ranker
$2^{18} \cdot S$	$\frac{S}{2^5}$	S	$\frac{13 \cdot S}{2^2}$	$S + \frac{1}{2^{18}}$	$2^2 \cdot D$

Table 4.3: Size in megabytes for each EXCES data structure. S and D are the ECD and the disk sizes in GB respectively. The Phase Table is a temporary data structure used only during reconfiguration.

4.3. The calculated sizes are for the worst case, i.e., when the ECD is completely filled of data. These formulas give us permanent memory usage of 0.1% and a temporary usage of 0.3% (during reconfiguration) relative to the ECD size. We believe these values are well-within acceptable limits.

To measure the CPU overhead due to EXCES, we used the following microbenchmark on the `beer` system. The microbenchmark issues a `grep` operation for a non-existent string on the EXCES source directory 100 times, that created a total of 21264 I/O operations. We divide the CPU overhead into two parts: the processing of the request before it is issued to the storage device (either ECD or disk), and the processing after the completion of the request. On an average, for each I/O operation, the corresponding numbers were 52 μs and 0.58 μs. Based on these, EXCES adds an average latency of 0.05 ms to the processing of each I/O request, which is relatively small compared to disk latency ($\geq$ 1 ms) and ECD latency ($\geq$ 0.5ms) [GF07]. While our current implementation of EXCES optimizes several operations, we believe that there is room for further improving this overhead time. Finally, we measured the reconfiguration overhead for the moderately I/O intensive BLTK developer benchmark. The average per-page reconfiguration time was measured to be 722 μs, an acceptable value for an infrequent operation.

4.7 Summary

In this chapter we present EXCES, an external caching system that reduces system energy consumption by prefetching, caching, and buffering disk data on a less energy consuming, persistent, external caching device. While external caching systems have been proposed in the past, EXCES is the first implementation and evaluation of such a system. We conducted a systematic evaluation of EXCES to determine overall energy savings and the impact on application performance. EXCES delivered overall system energy savings in the modest range of ∼2-14% across the BLTK and Postmark benchmarks. Further, we demonstrated that external caching systems can substantially impact application performance, especially for a write-intensive workload.

Note that the energy savings of EXCES and SRCMap (see §3) can be further improved by eliminating page *fetch-before-update* operations that would otherwise disturb spun down disks. Furthermore, we can also save energy by reducing the amount of memory required by the system, whereas the solutions that we have discussed so far focused on reducing energy utilization by the disk. In light of these observations, the next chapter presents non-blocking writes, a technique for eliminating *fetch-before-update* operations. Non-blocking writes can be used in conjunction with both SRCMap and EXCES, to improve the disk idle time without degrading the performance of the system. Furthermore, we explain how non-blocking writes has the potential to increase performance on systems with reduced memory to save energy.

4.8 Credits

EXCES was first published in the proceedings of the *IEEE International Symposium on High-Performance Computer Architecture* in February 2008 [UGB+08] and was presented by Luis Useche. Luis Useche contributed the initial design of EXCES including each component and their interactions. All the authors contributed in subsequent iterations of the system design. Luis Useche, Jorge Guerra, Mauricio Alarcon, and Medha Bhadkamkar implemented the system for the Linux kernel. Luis Useche and Jorge Guerra executed all the experiments to evaluate the system.

CHAPTER 5

CONTROLLING I/O TRAFFIC WITH NON-BLOCKING WRITES

Chapters 3 and 4 presented techniques for improving the energy efficiency of storage systems. These solutions achieve energy efficiency by replicating popular data and spinning down the primary data stores. The effectiveness of these techniques increases as spun-down disks are kept undisturbed for longer periods of time. The page *fetch-before-update* behavior in commodity operating systems can reduce the disk idle time for write operations. Hence, eliminating *fetch-before-update* operations offers a chance to keep disks idle for longer periods of time. In this chapter, we present the design, implementation, and evaluation of non-blocking writes, a solution for eliminating the *fetch-before-update* behavior. Non-blocking writes temporarily buffer write operations—while delaying fetch operations—and immediately returns control to the application. This technique can not only increase the energy efficiency of caching systems like EXCES and SRCMap but also improve the performance of low-memory but energy-efficient systems with high paging rates.

5.1 The Fetch-before-update Behavior

Writing data to main memory is not always fast. When using demand-paged virtual memory and file systems, the main memory caches a *subset* of the data contained within a backing store, typically a hard disk or SSD (array) device. However, memory references are done at a much smaller granularity (32 or 64 bit words) than is possible in backing stores. When data not present in memory is read or modified by a process, the operating system (OS) must fetch an entire page, typically 4KB or 8KB in commodity OSes, that contains the few bytes referenced by the processor. *Page fetches* occur in two scenarios: (1) a page mapped to the process address space (anonymous or file system page cache page) not resident in physical memory gets

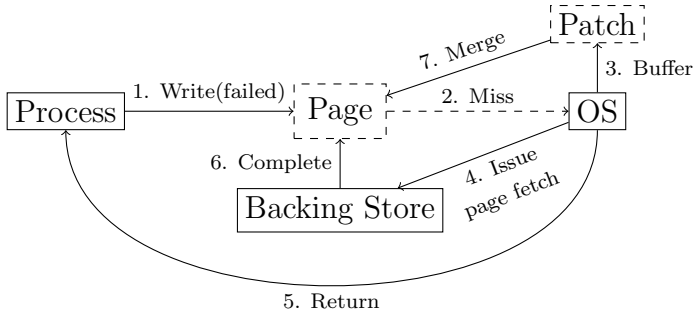


Figure 5.1: A non-blocking write in action. The first step, a write reference, fails because the page is not yet in memory. The dashed boxes are non-active entities.

accessed by a machine instruction (e.g., `LOAD` or `STORE`) causing a page fault, or (2) a system call accesses the OS page cache (e.g. `OPEN`, `READ`, or `WRITE`). Since a page fetch blocks the process, the performance of the running process during that time is limited by the performance of the backing store.

For read references, such blocking cannot be avoided since there is no other provision to correctly generate the data being read. Interestingly, the same blocking approach has been applied to handle write references in commodity operating systems and hypervisors till today (e.g., see recent Linux 2.6.34.5 and Xen 4.1.0 kernels). These include write references to swap-backed process memory or disk-backed file system pages, either directly (for anonymous and memory mapped pages) or via system calls (for OS cached pages). Thus, the writing process blocks in case the page being referenced is not in core memory until the referenced page is synchronously read from the backing store, leading to a *fetch-before-update* requirement [MBKQ96]. We demonstrate that writes *can* and *should* be handled differently and propose an approach to eliminate blocking for all write references to memory. We observe that in case of write references, instead of blocking the process to read in the page, the operating system can absorb such writes in temporary buffer pages and allow the process to continue executing immediately. The operating system can issue the page

read I/O to the backing store asynchronously and merge the update later when the page has been read into memory. A graphical representation of this process appears in Figure 5.1.

The proposed approach improves system performance in two ways. First, it immediately unblocks the process which is then free to execute subsequent instructions and make progress; the originally blocking page fetch operation can now overlap with useful computation. Second, it improves the parallelism of data retrieval from the backing store by creating the ability to keep many outstanding I/O (OIO) operations to handle multiple page fetches by a single process at the same time. Doing so leads to better throughput for both SSD and hard disk based backing stores, both of which offer better I/O throughput with multiple outstanding I/O operations.

On the energy side, eliminating the *fetch-before-update* has the potential to improve the effectiveness of current energy efficient systems. First, the gain in performance helps to improve their practical usability. For instance, systems that exchange DRAM with a larger but more energy efficient flash have a higher paging rate that can potentially benefit from delaying fetches when applications update pages. Second, for storage, the flexibility offered by non-blocking operations can save energy by serving page fetches in batches and increasing the disk idleness as opposed to individual requests interspersed by a very small amount of idleness.

Although our approach can be used to save energy, in this chapter we start with an early design focused on performance. The proposed design can be easily combined with minimal changes in any energy efficient system like SRCMap (see §3) or EXCES (see §4) and improve their effectiveness even further.

5.2 Motivating Non-blocking Writes

Technology trends indicate that page fetch rates are likely to increase in the future on many platforms. On the server end, multi-core systems and virtualization now enable more co-located workloads leading to larger memory working sets in systems. A recent report from VMWare indicated that of the four main computing resources for a typical system, the average utilization rates for memory are the highest (at 40%) compared to average utilization rates of less than 10% for the other resources [VMw]. On the personal computing end, increasingly data intensive desktop/laptop applications continue to place greater I/O demands [HDV+11]. Recent findings also show that page fetches and storage I/O also affect the performance of the increasingly data-intensive applications on mobile platforms significantly [KAU12]. Flash-based hybrid memory systems and storage caching and tiering systems are also motivated by these trends [CKZ11, GPG+11, SS10a, KM06, WR10]. This additional high-performance SSD layer in the storage stack combined with the increasingly data intensive nature of many workloads support a move to higher page fetch rates in future systems.

To better understand the potential impact of non-blocking writes, it is useful to evaluate the significance of the blocking page fetch problem in real-world workloads. We start by using a trace-driven virtual memory simulator that gives us an initial estimates of non-blocking fetches. Then, we use an instrumented Linux to measure the amount of time processes spent waiting for blocking page fetch operations as well as the fraction of page fetches due to writes that could be made non-blocking. We ran on both systems a broad range of workloads that exercise process anonymous memory as well as the file system page cache and present our analysis in this section. Additionally, we examine the alternate process execution model enabled

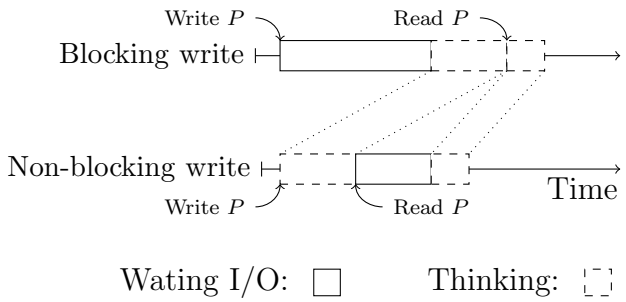


Figure 5.2: Page fetch asynchrony with non-blocking writes.. When the out-of-core page P is written, the application waits for the completion of the I/O. A brief thinktime is followed by a read to P. With non-blocking writes, since the write returns immediately, computation and I/O are performed in parallel.

by non-blocking writes to understand how it improves process execution and I/O performance.

5.2.1 Solution Impact

Non-blocking writes are designed to eliminate the *fetch-before-update* requirement. By eliminating *fetch-before-update*, non-blocking writes improves the performance by first, eliminating the synchronous page fetch latency for all out-of-core page writes and many out-of-core page reads, and second, parallelizing independent page fetches that would otherwise be serviced sequentially. While we discuss detailed design in the next section, we now consider how non-blocking writes alters process execution conceptually and quantify to what extent applications can benefit from them.

We show the potential for non-blocking writes for anonymous memory by analyzing two sets of benchmarks. Fist, we analyze a sub-set of the DaCapo benchmark suite as well as SPEC Power with our virtual memory simulator. Then, we study a subset of the SPEC CPU2006 benchmark suite designed to stress the memory subsystem, CPU, and compiler.

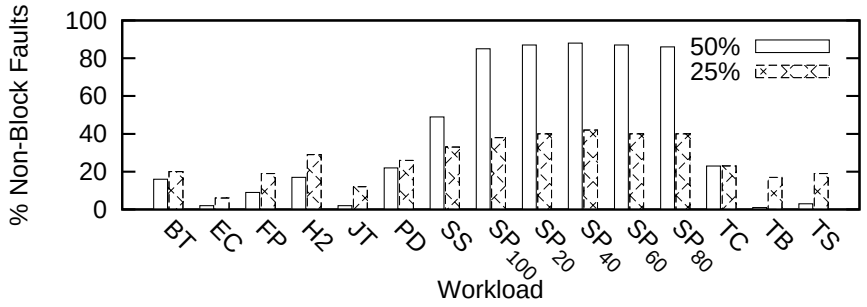


Figure 5.3: Fraction of page faults that benefit from non-blocking writes for various workloads with two memory sizes: 50% and 25% of each workload's footprint. We excluded those non-blocking writes that were followed by a non-blocking read.

	omnetpp	astar	gcc	xalancbmk		
Memory Provisioned (MB)	200	250	300	270	300	320
% Asynchronous Fetches	1.3	22.1	15.5	8.7	10.3	10.3
% Time Waiting I/O	5.9	28.1	37.5	70.9	47.7	30.5
% Possible Improvement	0.08	6.2	5.8	6.2	4.9	3.14

Table 5.1: Time benefit estimation for non-blocking writes. Two statistics to estimate the amount of time that can be saved with non-blocking writes for four benchmarks of the SPEC CPU2006 suite: (a) Fraction of page fetches that can be handled asynchronously and (b) Fraction of time spent by the benchmark waiting for I/O. The product of both quantities represents the expected time savings we could have for these benchmarks using non-blocking writes.

Page Fetch Asynchrony. With non-blocking writes, page fetches are made asynchronous. When an application writes to a page not available in memory, operating systems issue the read request, block, and return control to the application only after the page is fetched and updated. In the same situation, non-blocking writes buffers the written data in memory and returns control to the application, allowing the application to make progress immediately. Figure 5.2 depicts this improvement graphically. To have an initial estimation of possible asynchronous fetches, we fed our virtual memory simulator with full system memory traces of heterogeneous workloads summarized in Table 5.2. Figure 5.3 shows that with a DRAM provisioned for storing 50% of the total memory pages referenced, there is a substantial fraction—as much as 80%—of the total page faults (including both read and write faults) that can benefit from non-blocking writes for a variety of workloads. We also instrumented Linux to measure the fraction of page fetches that can be performed asynchronously with a full implementation of non-blocking writes. Table 5.1 shows this information for a subset of the SPEC CPU2006 benchmark workloads summarized in Table 5.3. In the case of xalancbmk, we also show the same information varying the amount of memory provisioned. We see a decrease in the number of page fetches that can be asynchronous when memory is reduced for xalancbmk. This indicates that most of the additional fetches are due to reads. Furthermore, we found that in the best case, non-blocking writes can make up to 22% of the major faults asynchronous.

The results for both benchmark suites, SPEC CPU2006 and DaCapo, were obtained using two different methods, real implementation and simulation. Despite their difference in how they were estimated, their results show similar number of possible asynchronous fetches, 13% and 19% on average respectively. Both benchmark suites are workloads to simulate real applications that exercise CPU and virtual

memory sub-system. Based on this evidence, we can conclude that it is common for computing and memory intensive applications to have a substantial amount of fetches that can be serviced asynchronously.

Preliminary Estimations. Applications can only benefits from asynchronous fetches if they spend a significant amount of time waiting for I/O. In our four SPEC CPU2006 benchmarks, we calculated the fraction of time each application is blocked while waiting for I/O completion. We show these results in Table 5.1. Since non-blocking writes decrease only the I/O waiting time and we know what fraction of the I/Os can be serviced asynchronously, we can calculate an expected improvement due to non-blocking writes by combining these two statistics. Table 5.1 shows the expected time savings for all benchmarks. Non-blocking writes will be able to save at most 6.2% of time from these benchmarks. Moreover, in the case of xalancbmk, we see that as the memory is reduced the potential benefits of non-blocking writes increases. This is due to the fact that the amount of time waiting for I/O increases when the memory is reduced. We report the actual results of running these benchmarks using non-blocking writes in Section 5.7.2.

Page Fetch Parallelism. Applications that access multiple pages not resident in memory during their execution are typically blocked by the operating system, once for each page while fetching it. Following this approach, operating systems end up sequentializing page fetches for accesses that are independent of each other. With non-blocking writes, the operating system is able to fetch pages in parallel taking better advantage of the typically available I/O parallelism at the device level. Higher levels of I/O parallelism typically lead to greater device I/O throughput which ultimately improves page fetch throughput for the application. Figure 5.4 depicts this improvement graphically.

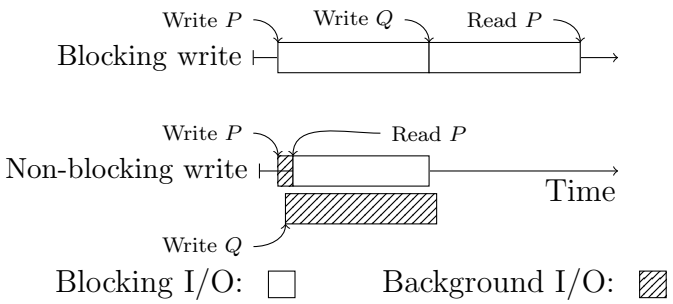


Figure 5.4: Page fetch parallelism with non-blocking writes. Two out-of-core pages, P and Q, are written in sequence and the page fetches get serialized by default. With non-blocking writes, P and Q get fetched in parallel increasing device I/O parallelism and thus page fetch throughput. Note that the read to P must still block until the page is fetched.

Shaping Resource Utilization. The page fetch and merge steps in non-blocking writes (Figure 5.1) are not necessary for continued process execution. As we shall discuss in §5.4, by delaying and/or intelligently scheduling page fetch operations, we can reduce and shape both memory consumption and the page fetch I/O traffic to storage due to out-of-core page accesses. These variant designs for non-blocking writes present previously unavailable knobs to shape application memory and storage resource consumption at a fundamental level. Such knobs can then be set to better match the dynamic availability of resources in the system.

5.3 Non-blocking Writes

Out-of-core page fetching is a central mechanism in commodity operating systems that enables demand paging for virtual memory and a file system page cache. When an out-of-core page is accessed either directly by the application or within the file system, the OS goes through a process of completing this access as depicted in Figure 5.5. In the *Check Page* state, it gathers all the information required to find the page and performs the look-up in memory first and, if necessary, external

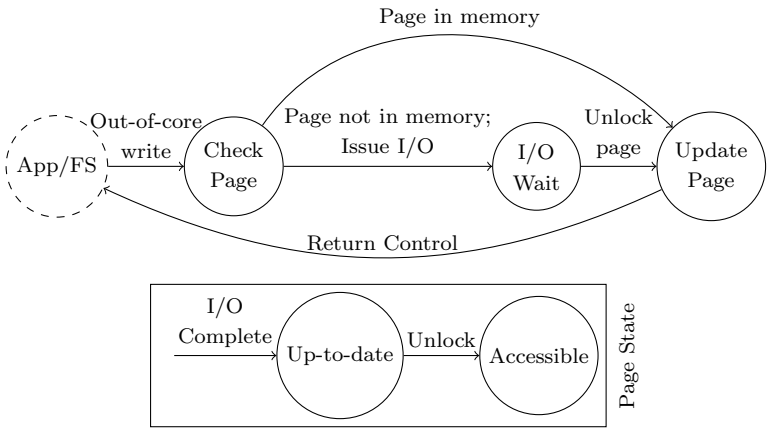


Figure 5.5: Process and page state diagram for out-of-core page access. The solid line states occur in kernel mode. The solid line rectangle frames the state of the page while the fault is handled.

storage. If the page is already in memory (as a result of recent loading), it jumps directly to the *Update Page* state. If the page is on the disk, it issues an I/O and waits for the I/O to complete and for the page to be set to the up-to-date state in memory (*Wait* state). When the I/O completes, the page is up-to-date and ready to be unlocked (states *Up-to-date* and *Accessible* of page state diagram). In the *Update Page* state, the OS sets up the page table (if applicable) and makes the page accessible. Finally, the control flow reverts to the original entity performing the out-of-core access.

In this section we present the design of non-blocking writes to eliminate the waiting state from the out-of-core page access processing. We present the challenges of non-blocking writes and their solutions along with new optimizations to further reduce the processing blocking.

5.3.1 Approach Overview

Non-blocking writes work by buffering updates to pages not available in memory. The basic approach modifies the out-of-core page fetch path as illustrated in Fig-

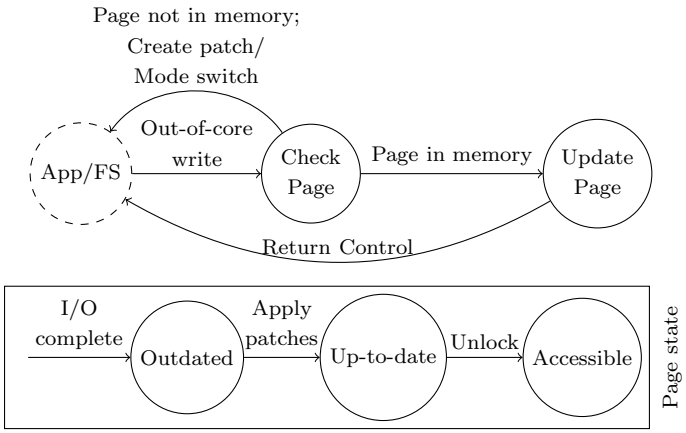


Figure 5.6: Process and page state diagram for out-of-core page access with non-blocking writes.

ure 5.6. In contrast to current systems, non-blocking writes eliminates the *I/O Wait* state that blocks the process until the page is available in memory. Instead, non-blocking writes returns immediately once a patch of the update is created and queued to the list of pending page updates. Additionally, non-blocking writes adds a new state in the page state, *Outdated*, that reflects the state of the page after it is read into memory but before pending patches are applied. The page transitions into the *Up-to-date* state once all the pending patches are applied. Within this framework, several questions must be addressed:

1. How should the OS extract the information required to populate each patch in the *Pre Fault* state?
2. How should the OS manage pending updates to an out-of-core page to ensure consistent data access?
3. How should the OS handle out-of-core page reads?

The rest of the section addresses these questions in the context of general systems design as well as implementations specific to Linux and the Intel architecture.

5.3.2 Write Interposition

Interposing on writes to out-of-core pages is a prerequisite for non-blocking writes. It enables recording of the data update to the page and subsequent unblocking of the process, thereby deviating from the normal control flow of the OS when handling out-of-core writes. Operating systems allow data writes via two common access interfaces: *memory mapped* and *system call*. Memory mapping allows backing up a portion of the process address space using either a file or the swap partition (anonymous memory). The processor can then write directly to memory without OS intervention (i.e., in *user-mode*). If the page is not available in memory at the moment of access, the processor generates a fault that is handled by the OS by fetching the page from its backing store. This operation is available in UNIX systems with the `mmap` system call. We call such writes *unsupervised*; the application writes to the memory page directly without requesting the OS.

Write access to files is also provided through system calls and this is the more common access path to filesystem data. In order to reduce the number of accesses to disk, the OS uses a buffer cache of file pages in memory from which the data is read in case it is available. When the accessed page is not cached, it is fetched from the backing store, updated, and stored in the OS buffer cache in anticipation of future accesses. We call such writes *supervised* and these are invoked using the *write* system call.

Supervised Writes

For handling supervised writes to an out-of-core page, the OS has all the information it needs to set up the non-blocking write. The system call arguments include the address of the data buffer to be written, the size of the data, and the file (and

implicitly, the offset) to write to. The OS resolves this to a page write internally and determines that the page is not cached.

In current systems, the OS allocates a page of memory to read in the out-of-core data. It then issues a blocking fetch of the out-of-core page, applies the update once the page is in memory, and only then unblocks the writing process. Contrarily, with non-blocking writes, the OS simply extracts the data update from the system call invocation using the address and size of the data buffer arguments, creates a patch, and queues it for later use. This patch is applied later when the data page is read into memory.

Unsupervised Writes

Handling unsupervised over-writes to an out-of-core page is substantially more involved. Modern ISAs (e.g., Intel) provide the reference address and the instruction that generated the page fault to the OS. Unfortunately, the amount of data written by the instruction generating the page fault varies across instructions and is not available directly. Moreover, the data written is not trivially obtained since the source could be a memory address, a CPU register, or a constant value, depending on the instruction.

Solution I. Our first solution combines partial disassembly with using a temporarily mapped page for single stepping the instruction to precisely extract the written data. This technique consists of the following steps:

1. First, we disassemble the instruction that caused the page fault. This step determines the number of bytes written which typically requires only a partial disassembly of the instruction for most CISC ISA's as well as RISC.
2. The faulting page is temporarily remapped to a free memory frame.

3. The faulting instruction is executed again using single stepping which will now successfully write to the temporarily mapped frame without faulting.
4. The data written by the single step execution is extracted from the temporarily mapped frame using the page offset in the faulting address provided by the processor and the size extracted in Step 1.
5. Once the single step is complete we restore the old page table entry to trap further faults on the original out-of-core page.

This approach is a general solution and will work for any architecture supporting the single stepping feature. It requires a minimum amount of information from the faulting instruction increasing the decoupling between this solution and the target ISA. Compared to page diff-merge, this solution requires only one additional execution of the faulting instruction and the space overhead is much lower as the temporary page can be freed once the data is extracted. On the downside, single stepping still requires two additional mode switches for every fault handled and a TLB flush to restore the original page table mapping. Both of these operations result in sub-optimal performance.

Solution II. In order to overcome the drawbacks of single stepping, we developed a solution based on full instruction disassembly to obtain the written data directly from the source, namely, register, memory, or constant value. Next, we simply skip the instruction that generated the fault and return control to the application. This solution does not require an additional buffer page nor does it incur a TLB flush every time a new page patch is created. However, it does require a full disassembly implementation for every ISA. If full disassembly is not practically feasible for specific instructions, the implementation could use partial disassembly with single-stepping or revert to blocking on the write without loss of correctness.

5.3.3 Page Patching

After patch information is extracted using write interposition, this information is temporarily saved. We now discuss how patch information is stored and applied to correctly update the page after it is fetched into memory.

Patch Creation

The data written needs to be stored along with additional information including the target location and size so that patches can be applied correctly upon page fetch. Since commodity operating systems handle data at the granularity of pages, we chose a design where each patch will apply to a single page. Thus, we abstract an update with a *page patch* data structure that contains all the information to patch and bring the page up-to-date.

To handle multiple disjoint overwrites to the same page, we implement *per-page patch queues* wherein page patches are queued and later applied to the page; adjacent patches get merged and overwrites to page locations overwrite the existing patch(es). It is important to note here that there is a one-to-one mapping of pages to physical memory frames in target environments (e.g., `struct page` in Linux or `struct vm_page` in OpenBSD) so memory sharing via page tables or otherwise is handled correctly. Consequently, shared pages share patch queues as well. Per-page patch queues also addresses consistency and ordering of updates to pages shared by several processes when using non-blocking writes. On operating systems with a unified buffer cache, this design decision also ensures consistency and ordering of page updates regardless if the write interposition occurred in memory mapped or file system page cache data. If two writes occur to an out-of-core page, one directly via memory mapping and the other to a file system managed page cache page via

the write system call, both patches will be queued and applied to the same page in the invocation order of the corresponding writes.

Patch Application

Patch application is rather straightforward. When a page is read in, first of all patches, if any, are applied 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. We then set the page flag indicating that the page is dirty (if any patches were applied) so that if the page needs to be swapped out it is correctly written to the backing store. Once all patches are applied, the page is unlocked which also unblocks the processes waiting on the page.

5.3.4 Non-blocking Reads

Reads to out-of-core pages block the process in current systems. However, with non-blocking writes, a new opportunity to perform *non-blocking reads* to out-of-core pages becomes available. Specifically, if the read is serviceable from one of the patches queued on the page, then the reading process can be unblocked immediately without having to block for a page fetch I/O. This occurs with no loss of correctness since the patch contains the most recent data written to the page.

Unsupervised reads are once again more challenging than supervised ones. For supervised reads, the page locations being read from and the target area to read into are both available as system call arguments. Since this is a read operation to a contiguous area, a simple lookup into the patch queue determines if the read is serviceable using the queued patches. The read is not serviceable if all data for the read is not contained within the patch queue and the reading process must block. If all data being requested is contained in the patch queue, the data is copied into

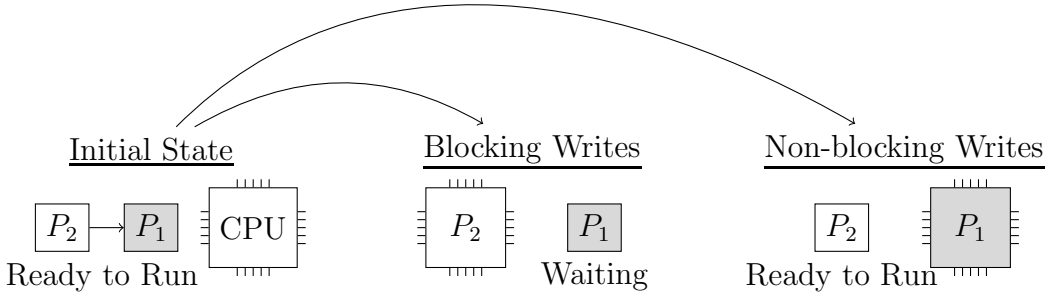


Figure 5.7: Example of non-blocking writes scheduling problem. This figure presents how the system would schedule the CPU when using blocking and non-blocking writes starting from the same initial state. The initial state has two processes ready to run, P_1 and P_2. P_1 is the next process to run. P_1 only writes to pages not in memory while P_2 progresses without operating system intervention.

the target buffer and the reading process is unblocked. Unsupervised reads are handled either using partial disassembly (to retrieve the source address size) and single-stepping via a temporarily mapped page or via full disassembly to additionally extract target memory are to copy the patch data into.

5.3.5 Scheduling with Non-blocking writes

Non-blocking writes is able to eliminate much of the I/O waiting time from applications. This leads processes to hold the CPU longer than they would normally do when writes to out-of-core pages block and lead the process to a context switch. An example of this behavior is graphically represented in Figure 5.7. We start with an initial state of two processes ready to run on a one CPU system. P_1 continuously generates faults by writing to pages not available in memory, while P_2 does not incur any faults. In the blocking writes configuration, after P_1 generates the first fault, the operating system will set it aside and schedule P_2 to use the CPU. In the non-blocking writes case, since P_1 can continue running while creating patches, it

continues to occupy the CPU until its time slot expires. Unfortunately, the progress of P_1 is inefficient given that every memory access requires kernel intervention. Meanwhile, P_2 is forced to wait for the CPU even though it would be able to operate faster as no additional overhead is incurred due to its memory accesses. In this example, the blocking writes configuration is more efficient given that it is forced to dedicate the CPU to the process that can progress faster. With non-blocking writes, the system is unable to detect the inefficiency of P_1 and ends up using the CPU to create patches while other processes waiting to be schedule can progress with no additional overhead.

Fortunately, the CPU scheduler can be redesigned to eliminate this problem by introducing a new process state. Figure 5.8 shows the current, as well as our proposed new process state diagram. We introduced a new process state called *NBW* (non-blocking writes) that applies to all processes currently creating patches. After a process write-faults on a page and before it creates its first patch, we move it to the *NBW* state and reschedule the CPU. A process can create patches for more than one page while it runs, hence, we need to keep track of all these pages in a per-page list we call G. We move a process from the *NBW* to the *ready to run* state when the I/Os to all pages in G are completed. The new scheduler will select processes to run in the *NBW* state only if there are no processes *ready to run*. Moreover, when a process is moved to the *ready to run* state, we make sure to reschedule the CPU if the running process is creating patches. The benefits of this new scheduling is two-fold. First, it solves the problem described in Figure 5.7 by giving priority to processes ready to run and have not created patches. Second, it guarantees that applications using non-blocking writes have, in the worst case, the same performance as current blocking writes while still exploiting unutilized CPU cycles when possible. The idea is that non-blocking writes is effectively moving some of the processes that were in

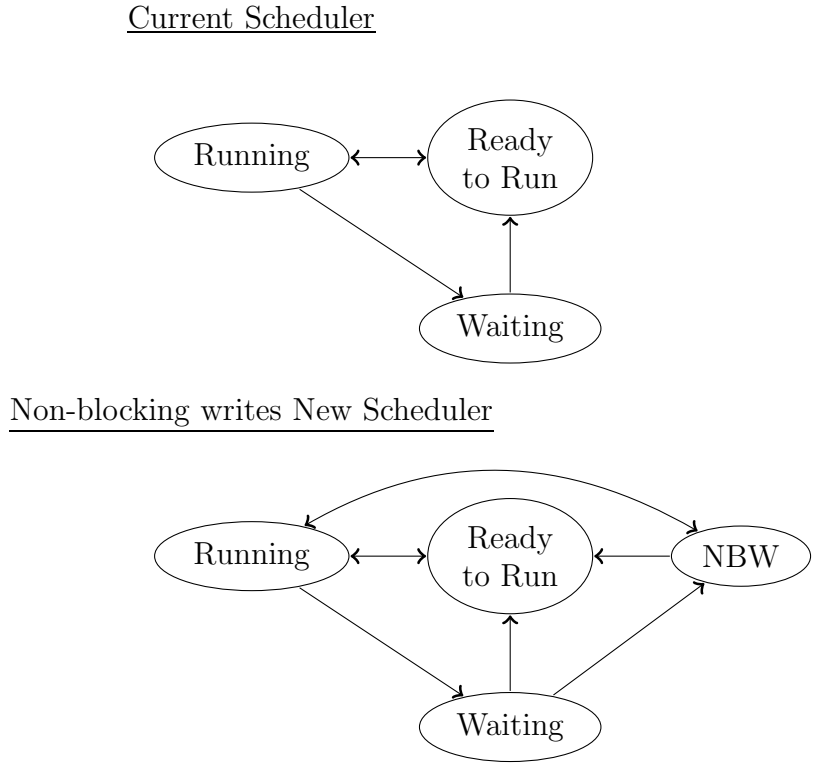


Figure 5.8: Current and new process state diagrams. The top figure shows the current process state diagram: ready to run, running, and waiting for I/O. The bottom figure shows the new proposed state diagram with one additional state (NBW) that applies to processes creating patches. When no process is ready to run, the scheduler can select processes in the NBW state to run in the CPU. When all pages for which a process was creating patches are fetched into memory, the process is moved to the ready state.

the waiting state to the new schedulable *NBW* state. Since we are running processes from this state only when there are no processes *ready to run*, we are using CPU cycles that would otherwise be wasted in the blocking writes configuration. This also ensures that processes in the *NBW* state do not use cycles that may be used by processes not creating patches.

5.4 Optimizations

Let us consider the page fetch operation issued in Step 4 when performing a non-blocking write as depicted in Figure 5.1. This operation requires both a *physical memory allocation* and a subsequent *asynchronous fetch* of the page so that the newly created patch and possibly subsequently created ones can be applied to the page. However, we note that since blocking is avoided, process execution is not dependent on the page being available in memory. This raises a key question: *if process execution is not contingent on the availability of the entire page, can page allocation and fetch be deferred or even eliminated?* Page fetch deferral and elimination are appealing because they allow reduction and shaping of both memory consumption and the page fetch I/O traffic to storage. Following this, we now explore optimizations of the basic approach to non-blocking writes presented in the previous section. These variants highlight the scope of possibilities that non-blocking writes enable which further optimize resource consumption and improve performance.

5.4.1 Alternative Page Fetching Modes

Asynchronous fetch as detailed in the previous section issues the page fetch I/O asynchronously before unblocking the writing process. The appeal of this approach is both in its simplicity and in the property that since the page is brought into

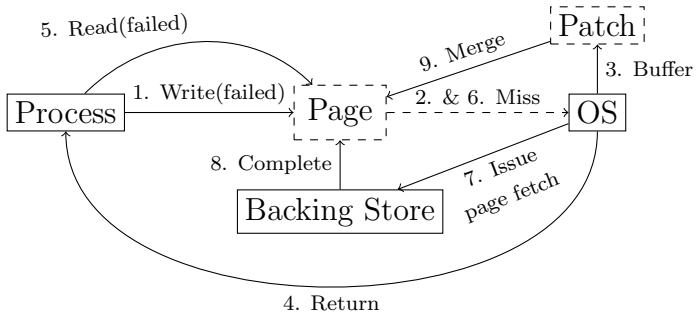


Figure 5.9: A non-blocking write with lazy fetch.. The Read operation in Step 5 optionally occurs after a time delay.

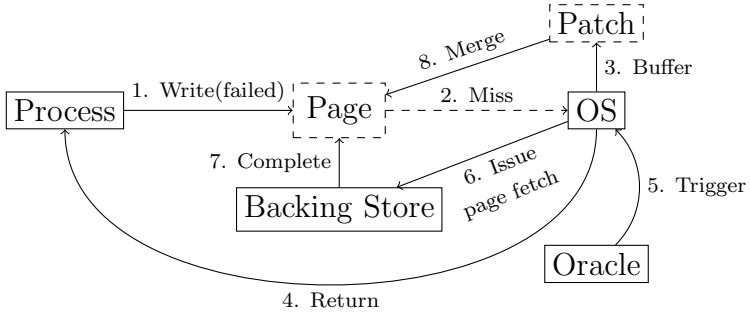


Figure 5.10: A non-blocking write with scheduled fetch.. An oracle triggers the page fetch operation.

memory in a timely fashion similar to the synchronous fetch, timer-based durability mechanisms such as dirty page flushing [Bac86] and file system journaling [Hag87] remain unaffected. However, asynchronously fetching pages immediately upon out-of-core write accesses indiscriminately uses system memory and storage I/O bandwidth both of which can be optimized. Relaxing this requirement of fetching pages immediately, new page fetching modes for non-blocking writes become possible.

Lazy Page Fetch

The first and obvious alternative is *lazy fetch* where the OS does not issue a page fetch I/O at all upon the out-of-core write access. Thus, the page would not be

fetched unless it becomes unavoidable as in cases discussed shortly. Figure 5.9 depicts this alternative graphically. Lazy fetch is an aggressive optimization which has the potential to further reduce the system's resource consumption in the correct circumstances. However, lazy fetch does not add any mechanism to flush patches not used recently. This could result in infrequently used patches taking memory that can potentially increase the major faults of the application as less space is available for new allocations.

Scheduled Page Fetch

Inspired by the shortcomings of lazy fetch, we designed a second alternative called *scheduled fetch* wherein the issuing of the page fetch I/O is scheduled at a later, more opportune time. Figure 5.10 depicts this alternative graphically. The design is similar to lazy fetch with the inclusion of a new component, *Oracle*, that dictates the policies of when to fetch the page. The policies enable better control over the system's resource consumption. Consequently, these page fetch policies should mitigate shortage of memory, high number of patch reads, or high load in the backing store.

First, to solve the shortage of memory, a thread can periodically scan the list of patch queues in memory in LRU order, and fetch the pages whose patches are using more memory than a given threshold. This threshold can be adjusted proportionally to the memory pressure currently present in the system.

Second, when an application is reading frequently from a patch, it is introducing all the kernel mode switch time in the total execution. We plan to mitigate this problem by fetching pages whose patches has been accessed repeatedly. The idea is to estimate the time spent by the application reading the same patch and issue the fetch once this time is equal to the average I/O time. This guarantees a maximum

degradation of performance of twice compared to fetching the page immediately after the patch is created.

Finally, in case where the memory is not under pressure and the backing store is under low-load, it is a convenient time to fetch the page. The load of the backing store can be calculated based on the number of outstanding I/Os it currently has. This will reduce the waiting time of the fetch as well as the memory usage while the I/O is completed.

5.4.2 To Fetch or Not to Fetch and When

The alternative page fetch modes create new scenarios under which page fetching must be evaluated. The first scenario occurs if a future page read cannot be served using the patches queued on the out-of-core page. The solution is obvious; since the page fetch is unavoidable, we fetch it synchronously and patches are applied first before the reading process is allowed to proceed with the read. The second scenario occurs if the page gets overwritten in its entirety (i.e., the patches created for the page are sufficient to fully reconstruct page data) and all page reads in the interim are satisfiable using the data contained in the patches associated with the page. In this scenario, if page durability is not necessary until the time the page gets entirely overwritten, the original page fetch is eliminated entirely. A third scenario occurs if there are no reads to the page or the page reads are satisfiable using the data contained in the patches associated with the page. In this scenario, if page durability is not necessary, the page fetch can be eliminated.

In scenarios 2 and 3 above, if page durability becomes a necessity at any point, the page is fetched synchronously at that point. Let us examine mechanisms in the OS that require data durability. Again, here we do not consider metadata durability

since non-blocking writes is not engaged for metadata pages and therefore they use the conventional durability mechanisms.

Data durability becomes necessary in the following instances: *(i)* memory reclamation by the virtual memory system [Tan07], *(ii)* synchronous file write by an application, and *(iii)* periodic flushing of dirty pages by the OS [Bac86], or page writes to a *write-ahead log* in a journaling file system [Hag87, PADAD05]), the The first case relates to process anonymous memory. Over time, it is possible with deferred and lazy fetching that patches to anonymous pages consume a substantial fraction of memory and many of these patches may relate to pages that are not in active use. Under memory pressure, for the virtual memory system to be able to reclaim the memory used to store such patches, the modifications contained within those patches must be made durable for correctness since these pages *may* become actively used again at some later point. The second and third cases relates to file system buffer cache data durability operations initiated synchronously by the application and by the file system respectively. In both of these cases, the page is fetched synchronously before being flushed to disk with no loss of correctness.

5.5 Correctness

Non-blocking writes alters the behavior and control flow of current systems. We now discuss how it preserves semantic correctness despite this change.

OS-initiated page accesses. Our current design does not implement non-blocking writes for all accesses (writes and reads) to out-of-core memory pages that are initiated internally by the OS. These include file system metadata page updates, and updates performed by kernel threads (e.g., the `bdflush` dirty page flushing thread and the `kjournald` journaling thread for the ext3 file system in Linux). For instance, when a journaling thread writes a file data page to storage, the thread is

blocked until it is first read into memory (if not present), updated by merging any pending patches, and only then unblocked to write the page out to storage. This design decision trivially provides the durability properties expected by OS services to preserve semantic correctness.

Synchronizing page operations. From the moment a non-blocking write operation starts and until it finishes, multiple operations like read, prefetching, synchronous write, and flush can be issued to the page. Operating systems need to synchronize these operations to keep the consistency and return only up-to-date data to applications. We achieve synchronization trivially by complying with the locking mechanism already existent within the operating system. Before setting the page as a non-blocking write page, we lock and setup the page just as before any other operation. This will block other operations as they would normally do until the data is up-to-date, i.e., the page is fetched into memory and patches applied. The only exception to this mechanism lies in the operation of writing to a page already in the non-blocking write state. In this case, we do not lock the page to queue a new patch as it was locked when the non-blocking write was first setup. In our implementation we comply with the Linux page locking protocol. First, we index the new allocated page in the page cache tree to make it public to other kernel subsystems. Second, we lock the page to block other operations until all the patches are applied.

Multiple pages. When distinct memory pages get written to by two processes (or threads), current operating systems do not provide ordering guarantees and neither does non-blocking writes. However, when distinct memory pages get written to by the same process sequentially, operating systems ensure that the updates to these pages occur in the correct sequence. If the first page written to is not in memory, the process is blocked to fetch it in first before allowing it to proceed on to writing the

second page. With non-blocking writes, in contrast, the second update can indeed occur prior to the first one. Such alternate ordering can also occur in the case when neither page is in memory and the fetch for the second page completes earlier than that of the first.

Alternate ordering as described above does not affect correctness. First, for all practical purposes, the creation of a patch constitutes updating the page in memory since processes reading from these locations will always see the most up-to-date data for either page (due to reading from patches) regardless of whether the page is in memory or not; thus, control flow dependent on the sequence of these updates is not altered at all. Second, these writes are to memory and are not guaranteed to be reflected to persistent storage in any particular sequence; therefore, the ordering violations are crash-safe. If a process would like explicit disk ordering for these memory page updates, the process would execute blocking flush operation (e.g., `fsync`) subsequent to each operation. The flush operation would cause the OS to wait for the page fetch and apply any outstanding patches before flushing and returning control to the application; ordering of disk writes would thus be preserved with non-blocking writes.

Handling of disk errors Non-blocking writes changes the semantics of the OS with respect to notification of errors to a process that writes to an out-of-core page. Since non-blocking writes performs page fetches asynchronously, disk I/O errors (e.g., `EIO` returned for the UNIX write system call) during the asynchronous page fetch operation would not get reported to the process. If the application were to take differential action under such states, such action may be engaged with a delay or not at all. We believe that this is just different semantics for the write system calls and not an actual *error*. Semantically speaking, the write itself was not made to persistent storage and only to memory and therefore the write was not in error

Workload	ID	Footprint (MB)	Exec. Time (secs)	# References ($\times 10^6$)
batik	BT	149	17.60	25
eclipse	EC	223	7.20	11
fop	FP	144	11.16	32
h2	H2	722	44.19	386
jython	JT	540	49.68	128
pmd	PD	170	20.86	60
tomcat	TC	215	33.39	118
tradebeans	TB	337	23.84	87
tradesoap	TS	335	31.56	84
postmark	SS	256	9.9	69
specpower-20	SP_{20}	218	22	44
specpower-40	SP_{40}	224	22	48
specpower-60	SP_{60}	214	22	53
specpower-80	SP_{80}	214	22	60
specpower-100	SP_{100}	211	22	63

Table 5.2: Workloads include a mix of DaCapo benchmark suite 9.12 [Bla06], PostMark [Kat97], and SPECpower [Lan09]. *SP_X X indicates the percentage of load in the system. PM is set to the small-small configuration used by Riska et al.[RLLR07].*

in the first place; the reporting of a disk I/O error (such as `EIO`) when the semantics guarantee only a write to memory is convoluted as well. More importantly, if the write were to be made persistent at any point via a flush issued by the process or the OS, any I/O errors during page flushing would get reported to the user of the system. Therefore, we believe that this change in semantics does not introduce a consistency violation.

5.6 Estimating Benefits

In this section, we quantify the potential benefits of non-blocking writes using a virtual memory simulator along with full-system memory traces of several heterogeneous workloads.

5.6.1 Virtual Memory Simulation

We built a virtual memory simulator that given a memory size and a memory trace, simulates hits, misses, and evictions of memory pages. As input, we employed full system memory traces of heterogeneous workloads summarized in Table 5.2. On every memory reference, the simulator reports the timestamp, operation mode (read or write), and event (hit, miss, or evict). More importantly, the simulator is designed to report the number of write faults that can benefit from non-blocking writes. To do so, the simulator must *(i)* be able to distinguish *over-writes* from *allocation-writes*, and *(ii)* determine which write fetches can really benefit from non-blocking writes.

Allocation-writes do not trigger I/O operations while over-writes may. Unfortunately, we do not have sufficient information in our traces to entirely distinguish one from the other. In order to minimize the occurrence of false positives when detecting over-writes, we use two heuristics. First, we consider the first write access to every page in the trace conservatively as an allocation write by default. Second, we use both the virtual and physical addresses to uniquely identify a page, instead of just the virtual address or the physical address alone. This eliminates false positives in detecting overwrites when the same virtual address is reused to map to a different physical address or vice-versa.

Finally, the simulator also employs a model to predict I/O latency for various values of OIO which is then used to determine when an asynchronous read related to a non-blocking write would complete. Using an approximation proposed by Gulati *et al.* that latency varies linearly with OIO [GKAK10] and training this model on a few points with a real SSD (PCIe OCZ Revodrive 160GB), we were able to predict the latency of the device for any arbitrary OIO value.

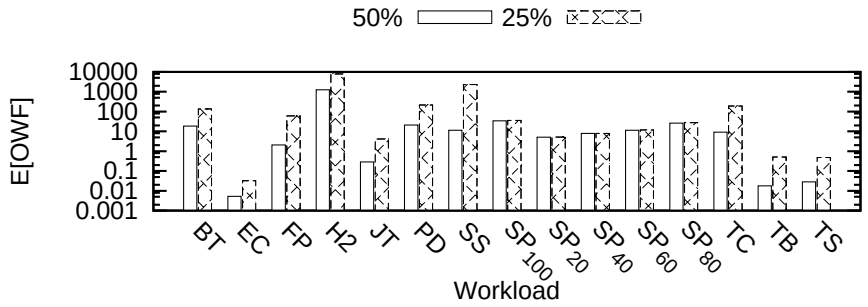


Figure 5.11: Expected OWF for various workloads with two different memory sizes: 50% and 25% of workload footprint.

5.6.2 Fraction of Non-blocking Write Faults

We measured the fraction of page faults that benefit from non-blocking writes for all workloads in Table 5.2. Figure 5.3 shows these results. When the provisioned DRAM is half of the total memory footprint (total size of memory referenced) for each workload, 2-88% of the faults are write fetches that benefit from non-blocking writes. When the provisioned DRAM is reduced to a fourth of the memory footprints, the fraction of non-blocking write faults is 7-42%. This implies that there is substantial potential for improving the overall performance of page fault related work for many of these workloads. For some workloads, a reduction in memory size causes a reduction in the percentage of non-blocking writes. For these workloads, higher page fault rates (as reduced memory sizes) lead to more of the pages involved in non-blocking writes being selected for eviction. Our simulator correctly revokes the non-blocking write status for such pages given that they must now block for the completion of the background read before they can be evicted.

5.6.3 Outstanding Write Fetches

While the fraction of non-blocking write faults are partially indicative, they do not suggest how the absolute number of simultaneous non-blocking writes vary over

time. To address this consideration, we define the *outstanding write fetches* (OWF) as the number of write faults that can still benefit from non-blocking writes at any time during the execution of a process. OWF gets incremented each time there is a write to an out-of-core page. An asynchronous read to the page is also initiated at that time and the page is marked as *in-io*. When there is a read reference by the process to an *in-io* page, or an *in-io* page is evicted, or if the simulated asynchronous read due to an *in-io* page completes, the OWF value gets decremented.

Figure 5.11 reports the expected value (time-weighted average) of the OWF for each of the workloads when the DRAM size is configured to be 50% or 25% of the total memory footprint. First, we note that all workloads show values greater than zero, indicating that each can benefit from non-blocking writes. For most workloads, the OWF ranges from 2 to 33, indicating a healthy opportunity for parallelizing the asynchronous reads due to non-blocking writes. The two exceptions are EC, which has the lowest OWF value (0.005), and H2, which has an exceptionally high OWF value (1259.07).

5.6.4 Estimating Overall Savings

To estimate how non-blocking writes would impact execution times for the DaCapo workloads, we revisit the percentage of page fetches that can benefit from non-blocking writes (discussed earlier in Figure 5.3). This gives an upper bound on the savings in running time that our set of benchmarks could have with non-blocking writes. With 50% DRAM provisioning relative to memory footprint size, half of the workloads show 20% or less page fetches that benefit from non-blocking writes. However, the range is 30% to 80% for the remaining half which incur more paging activity. Combining this finding with those from previous studies which show that applications using paging and are heavily optimized spend more than 40% on disk

I/O [SS10a], we could estimate an overall reduction of 12% to 32% in application execution times for these workloads.

5.7 Evaluation

In this section, we describe an evaluation of our implementation of *asynchronous* page fetch variant of non-blocking writes. We did not evaluate *lazy* and *scheduled* page fetch variants to focus deeper on the asynchronous fetch mode. We will evaluate lazy and scheduled fetch as a second iteration over the design and implementation of non-blocking writes. For performance evaluation, we used a subset of the DaCapo, SPEC CPU2006, SPEC Power2008, and SPEC SFS2008 benchmark suites [Sta]. The SPEC CPU subset was chosen to exclude floating point workloads (our disassembly engine does not handle floating point instructions currently) and benchmarks with small memory footprints (< 200MB). We ran the (subsets) DaCapo, SPEC CPU2006 and SPEC Power2008 benchmarks on a Dell PowerEdge T105 with a single Quad-Core AMD Opteron 2.3GHz and an Intel SSD with 160GB of storage. The SPEC SFS2008 benchmark were run on the same hardware except we used a 500GB SATA hard drive for better representation of a real file server.

5.7.1 Experimental setup

We implemented non-blocking writes in the Linux kernel 2.6.34.5. Our implementation includes the following features:

- Full instruction disassembly for unsupervised writes.
- Single-stepping with temporal buffer page when instruction disassembly is not possible.

Workload	Working-set Size (GB)	Memory Provisioned (MB)	Workload Type
speccpu-omnetpp	0.6	200	memory
speccpu-astar	1.3	250	memory
speccpu-gcc	3.5	300	memory
speccpu-xalancbmk	1.6	300	memory
specsfs	12	512	filesystem
batik	0.14	72	memory
fop	0.14	72	memory
pmd	0.17	85	memory
h2	0.71	384	memory

Table 5.3: Mix of heterogeneous workloads to evaluate non-blocking writes.

- Patch merging when writing to adjacent locations or overwriting existing patches.
- Non-blocking reads when data is available in patches.

To simplify our implementation, we disabled SMP and kernel preemption in the kernel configuration. We ran Gentoo Linux on top of our customized kernel. We used workloads that stress both the virtual memory and the file system. These workloads range from interpreters and compilers to file and application servers. To exercise non-blocking writes, we configured the memory size for each workload so that it incurs I/O activity. We calculated this value by starting with a memory equal to the workload's working-set and progressively decreased its size until the system started paging. Table 5.3 summarize the workloads we used along with its working-set sizes and the amount of physical memory provisioned for each. The type of workload indicates if the benchmark exercises the virtual memory subsystem or the file system path.

We ran the SPEC CPU2006 benchmarks with the *ref* dataset, its representation of a real workload for five iterations. We set up SPEC SFS2008 with a target of 100 operations per second and noted the response time from the NFS server,

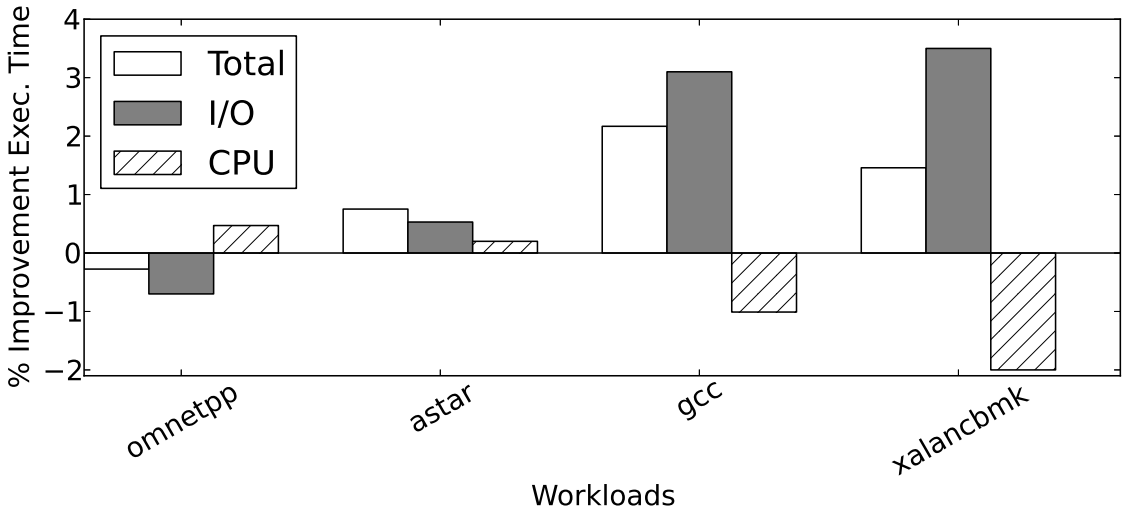


Figure 5.12: Execution time change in single-threaded applications. Percentage of improvement in execution time separated by total, I/O waiting, and CPU.

mounting the file system in *async* mode in each case. SPEC Power2008 ran its default three adjustment rounds and one final round with 100% load which is the one we evaluated. We provisioned the DaCapo benchmarks with 50% of memory to be able to compare with our virtual simulator results. We evaluated performance when running default and non-blocking writes kernels for each workload.

5.7.2 Performance Improvements

We measured the performance improvements due to non-blocking writes along three dimensions: execution time, latency, and throughput. For the SPEC CPU2006 benchmarks, we measured the improvements in average execution time of five runs and reported these in the white bars of Figure 5.12. In all the experiments, the coefficient of variation was below 2%. The average improvement in execution time across these benchmarks was 1%, with as much as 2.1% reduction in the best case, and a degradation of less than 0.3% in the worst case.

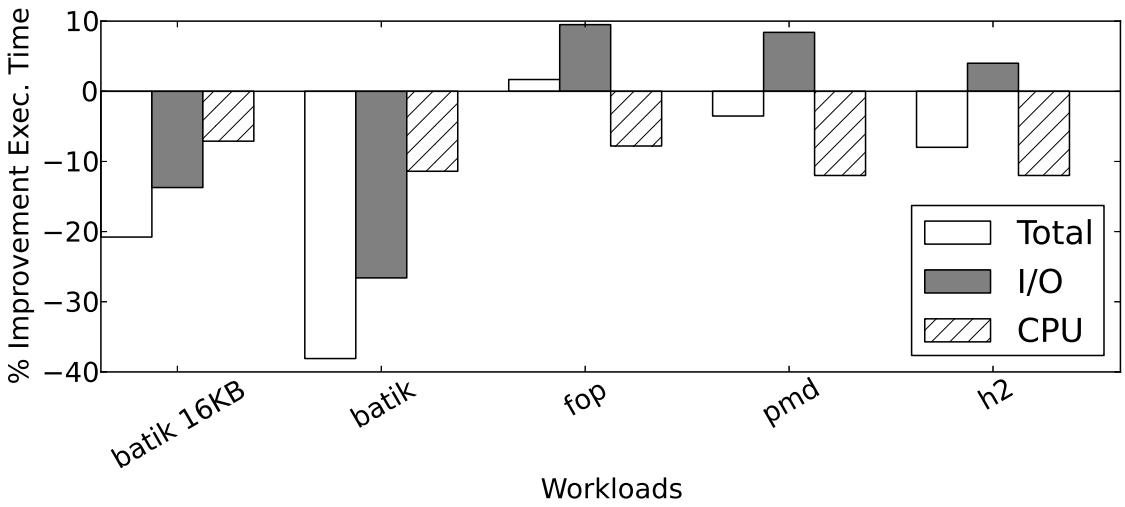


Figure 5.13: Execution time change in multi-threaded applications. Percentage of improvement in execution time separated by total, I/O waiting, and CPU.

To understand better the source of improvements, we separated CPU time from I/O waiting time as shown in the gray and striped bars of Figure 5.12. As expected, non-blocking writes is able to reduce the time that the application is waiting for I/O. However, we also see an increase in the CPU time of the application. The reason for this is that, while current systems blocks when accessing an out-of-core page, non-blocking writes is able to stay active creating patches for pages not available in memory. In particular, workloads that create long queues of patches do worse in CPU time since it takes more time to perform the writes compared to blocking writes when the page is in memory. Note that this increase in computation time may also increases the amount of energy used by the CPU. While in use, the CPU is not able to transition into a energy saving state. In the future, we plan to compare the energy efficiency achieved by non-blocking writes with the additional energy consumed by the CPU and check if we still have a positive balance in energy savings.

All the SPEC CPU2006 benchmarks are single-threaded. We expanded our evaluation by exercising non-blocking writes with the DaCapo multi-threaded applica-

tions. We picked four of the benchmarks presented in Table 5.2: batik, fop, pmd, and h2. Similarly to the SPEC CPU2006 experiments, we measured the improvement in average execution time of five runs. We report the results in Figure 5.13 separating the change in time into I/O and CPU. Due to their non-deterministic nature, these results have a large variance, with the coefficient of variation reaching 25% in the worst case. In contrast with the predicted improvements in Section 5.6, non-blocking writes was not able to improve the performance for all benchmarks. In all cases, except batik, non-blocking writes reduces the time that the application has to wait for I/O. However, non-blocking writes increments the CPU time considerably. In fact, the increase in CPU time surpasses the benefits obtained by waiting less time for I/O. We believe this is the result of sub-optimal scheduling of the resources when using non-blocking writes with multi-threaded applications and this only occurs on non-blocking writes operations for memory mapped accesses. In Section 5.3.5 we detail the problem and present a new CPU scheduling technique that, in the worst case, guarantees applications the same performance as current blocking writes.

Finally, we see an anomaly in the behavior of batik. For this benchmark, non-blocking writes not only uses more CPU, but it also increases the time the application is waiting for I/O. For this application, we observed an increase of 27% in the number of major faults. Although we found that non-blocking writes with batik uses on average 15KB of memory to store patch data, we ran additional experiments where we limit the memory used by non-blocking writes to a maximum of 16KB and eliminate potential issues with peak memory usage. The results are in Figure 5.13 under the *batik 16KB* workload name. Although the new times decreased, non-blocking writes still increase the time spent waiting for I/O in *batik 16KB*. We observed that this behavior occurred only in one of the ten workloads we tested with

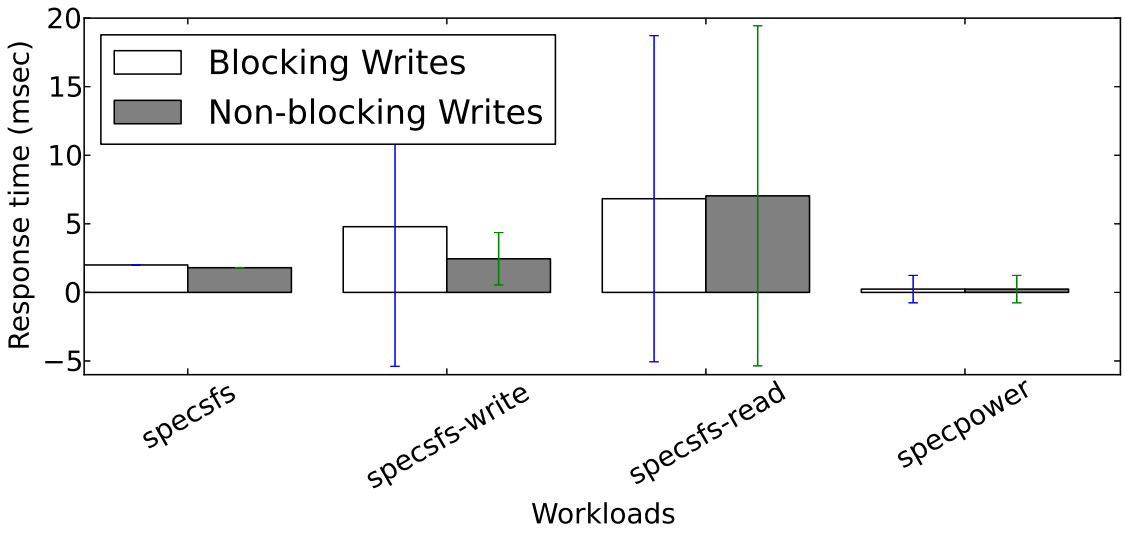


Figure 5.14: SPEC SFS2008 and SPEC Power2008 response times.

non-blocking writes. In the future, we plan to investigate further the source of the time increase waiting for I/O in this particular benchmark.

Next, we evaluated non-blocking writes in the context of file system workloads. We measured and reported the response time latency for the SPEC SFS2008 and SPEC Power2008. SPEC SFS2008 uses an NFS client-server configuration. We configured the server end to optionally use either the non-blocking writes or the default (Vanilla) kernel. SPEC Power2008 runs at a target throughput level specified by the user and we measured improvement in response time. Figure 5.14 shows the improvements in response time (latency) across these workloads. As expected, non-blocking writes is able to reduce the average write latency of SPEC SFS2008 by 50% while reads are mostly unchanged. Furthermore, non-blocking writes is able to decrease the variation in latency as writes are now less dependent on disk.

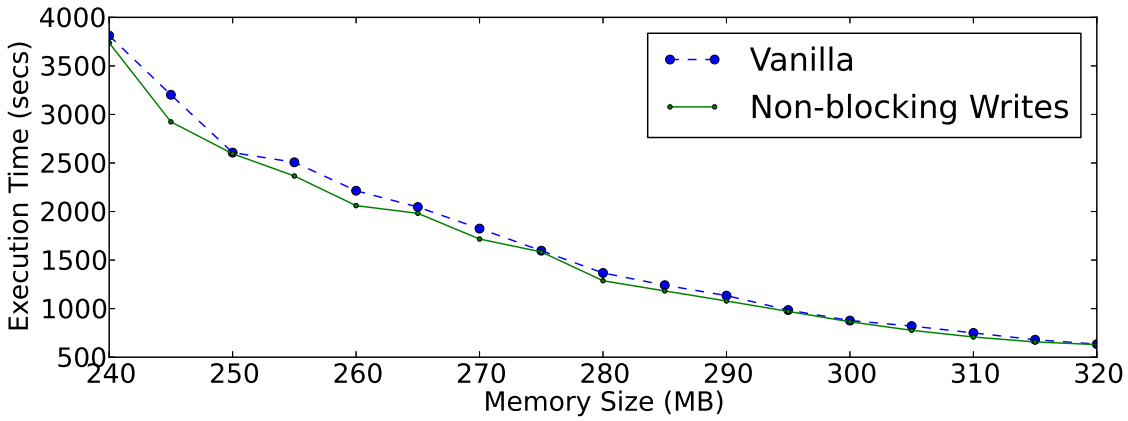


Figure 5.15: Sensitivity of performance due to non-blocking writes to the amount of memory available in the systems. We used the xalancbmk benchmark from the SPEC CPU2006 suite.

5.7.3 Memory Sensitivity

To check the effectiveness of non-blocking writes with memory availability, we made a set of experiments varying memory size, from 240MB to 320MB in 5MB increments, and noted the execution time for the *xalancbmk* SPEC CPU2006 benchmark.

Figure 5.15 depicts the execution time when varying memory for blocking as well as non-blocking writes. First, as expected, non-blocking writes has its lowest benefit—0.76% time improvement—when memory is well provisioned at 320MB. This is due to the fact that, as more memory is available, less *fetch-before-update* operations are required. In contrast, when the memory is set to 250MB, non-blocking writes shows its maximum benefits—8.7% time improvement—due to an increased chance to make more fetches asynchronous as more *fetch-before-update* operations are needed. It is clear from these experiments that non-blocking writes shows more benefits as less memory is available. This makes non-blocking writes a good candidate to improve the practical usability of systems with high paging rate that exchange DRAM with a large but more energy efficient flash.

Optimization	Exec. Time (sec)	% Improvement	% Occurence
No Optmization	2340.7	–	4.8
Patch Read	2233.1	4.6	2.5
Patch Merging	2229.6	7.7	92.7
Patch Read + Merging	2060.9	12	95.2

Table 5.4: Performance improvements due to patch optimizations. Four experiments of *xalancbmk* with memory provisioned at 260MB varying the patch optimizations. Each experiment enables only the optimization being evaluated. For each experiment, the table shows its execution time, its improvement compared to the no optimizations configuration, and percentage of occurrences when they are all enabled relative to the total non-blocking reads and writes.

5.7.4 Optimizations with Patches

We implemented two optimizations directly related to patches in non-blocking writes. First, we implemented the non-blocking read operations as explained in §5.3.4. Second, to reduce meta-data memory, we also implemented an optimization that merges updates that are adjacent or overwrite patches already in queue.

To understand the benefits of each of these optimizations, we ran four experiments of the benchmark *xalancbmk* with memory provisioned at 260MB varying the optimizations we used. We show the results in Table 5.4. We found the highest benefit of 7.7% for the patch merging optimization. Moreover, patch read alone improves execution time by 4.6%. Additionally, we found that 92.7% of all patches created were merged with a previous patch in the queue. When used in conjunction, all the optimizations accounted for a total of 12% decrease in the execution time of the benchmark when compared to non-blocking writes without these patch optimizations.

5.8 Summary

Since their original design, operating systems have blocked processes that write to out-of-core pages—both in file system and virtual memory—while fetch operations are being performed. In this chapter, we revisited the well-established design of the write primitive and demonstrated that such blocking is not just unnecessary but also detrimental to performance.

Our solution, non-blocking writes, decouples the process of writing data to a page from its presence in memory by buffering page updates and merging them later asynchronously once the page is fetched into memory. It achieves this decoupling with a self-contained operating system improvement that is transparent to the application and preserves semantic correctness. We proposed *asynchronous*, *lazy*, and *scheduled* page fetch variants, each design intended to progressively improve upon the previous.

We implemented non-blocking writes with a basic *asynchronous* page fetch design in Linux for x86 processors with very encouraging results. Our evaluation using single-threaded memory and file system intensive workloads revealed performance improvements of up to 8.7% in terms of execution time and a reduction of 50% in the average write response time in a file-server workload. However, when evaluated with multi-threaded benchmarks, non-blocking writes showed a degradation in execution time. We believe this is due to sub-optimal scheduling of the CPU. Threads creating patches have a slower rate of progress than those that do not require non-blocking writes to continue execution. Since non-blocking writes eliminates blocking due to I/O, these threads can occupy the CPU for long periods of time preventing processes with potential to progress faster from running. We proposed a CPU scheduler optimization to eliminate this problem by preferentially scheduling threads that do

not create patches. Moreover, this optimization guarantees that the performance will be comparable to the blocking writes configuration in the worst case.

We also tested the sensitivity of non-blocking writes to the amount of available memory in the systems with a set of experiment of SPEC CPU2006 xalancbmk. We found that the benefits of non-blocking writes increase as memory is reduced, up to 8.7% in execution time improvement when the memory is provisioned at 250MB. This demonstrates the potential of non-blocking writes to improve the practical usability of energy efficient systems with high paging rates.

The following chapter discusses other energy efficient systems from the literature, as well as additional existing works that explore methods for achieving asynchronous fetches that can also reduce the *fetch-before-update* I/O operations.

5.9 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, 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, Ricardo Koller, and Jesus Ramos implemented non-blocking writes for file system and memory mapped writes on the Linux kernel. Luis Useche designed and executed the experiments to evaluate the implementation of non-blocking writes.

CHAPTER 6

RELATED WORK

There is a large body of work in systems energy optimization as well as possible performance optimization that could mitigate the *fetch-before-update* problem. We divided the related work in three main categories: energy proportionality storage systems previous work, flash caching for energy efficient storage systems previous work, and possible solutions to mitigate the *fetch-before-update* problem.

6.1 Energy Proportionality in Storage Systems

It has been shown that the idleness in storage workload is quite low for typical server workloads [ZCT+05]. We examine several classes of related work that represent approaches to increase this idleness for energy minimization and evaluate the extent to which they address our design goals. We next discuss each of them and summarize their relative strengths in Table 6.1.

Singly redundant schemes The central idea used by these schemes is spinning down disks with redundant data during periods of low I/O load [GLM+08, PBD06, WYZ08]. RIMAC [WYZ08] uses memory-level and on-disk redundancy to reduce passive spin ups in RAID5 systems, enabling the spinning down of one out of the N disks in the array. The Diverted Accesses technique [PBD06] generalizes this

Design Goal	**Write offloading**	**Caching systems**	**Singly Redundant**	**Geared RAID**
Proportionality	∼	×	×	∼
Space overhead	✓	✓	×	×
Reliability	×	×	✓	✓
Adaptation	×	✓	✓	✓
Heterogeneity	∼	∼	∼	×

Table 6.1: Comparison of Power Management Techniques. ∼ indicates the goal is partially addressed.

approach to find the best redundancy configuration for energy, performance, and reliability for all RAID levels. Greenan *et al.* propose generic techniques for managing power-aware erasure coded storage systems [GLM+08]. The above techniques aim to support two energy levels and do not address fine-grained energy proportionality.

Geared RAIDs PARAID [WOQ+07] is a gear-shifting mechanism (each disk spun down represents a gear shift) for a parity-based RAID. To implement $N-1$ gears in a N disk array with used storage X, PARAID requires $O(X \log N)$ space, even if we ignore the space required for storing parity information. DiskGroup [LVW07] is a modification of RAID-1 that enables a subset of the disks in a mirror group to be activated as necessary. Both techniques incur large space overhead. Further, they do not address heterogeneous storage systems composed of multiple volumes with varying I/O workload intensities.

Caching systems This class of work is mostly based on caching popular data on additional storage [CG02, LLN08, UGB+08] to spin down primary data drives. MAID [CG02], an archival storage system, optionally uses additional cache disks for replicating popular data to increase idle periods on the remaining disks. PDC [PB04] does not use additional disks but rather suggests migrating data between disks according to popularity, always keeping the most popular data on a few active disks. EXCES [UGB+08] uses a low-end flash device for caching popular data and buffering writes to increase idle periods of disk drives. Lee *et al.* [LLN08] suggest augmenting RAID systems with an SSD for a similar purpose. A dedicated storage cache does not provide fine-grained energy proportionality; the storage system is able to save energy only when the I/O load is low and can be served from the cache.

Further, these techniques do not account for the reliability impact of frequent disk spin-up operations.

Write Offloading Write off-loading is an energy saving technique based on redirecting writes to alternate locations. The authors of write-offloading demonstrate that idle periods at a one minute granularity can be significantly increased by off-loading writes to a different volume. The reliability impact due to frequent spin-up cycles on a disk is a potential concern, which the authors acknowledge but leave as an open problem. In contrast, SRCMap increases the idle periods substantially by off-loading popular data reads in addition to the writes, and thus more comprehensively addressing this important concern. Another important question not addressed in the write off-loading work is: with multiple volumes, which active volume should be treated as a write off-loading target for each spun down volume? SRCMap addresses this question clearly with a formal process for identifying the set of active disks during each interval.

Other techniques There are orthogonal classes of work that can either be used in conjunction with SRCMap or that address other target environments. Hibernator [ZCT+05] uses DRPM [GSKF03] to create a multi-tier hierarchy of futuristic multi-speed disks. The speed for each disk is set and data migrated across tiers as the workload changes. Pergamum is an archival storage system designed to be energy-efficient with techniques for reducing inter-disk dependencies and staggering rebuild operations [SGMV08]. Gurumurthi *et al.* propose intra-disk parallelism on high capacity drives to improve disk bandwidth without increasing power consumption [GSS09]. Finally, Ganesh *et al.* propose log-structured striped writing on a disk array to increase the predictability of active/inactive spindles [GWBB07].

6.2 Energy Efficient Storage with Flash

We classify research related to EXCES into three categories: energy-saving external caching techniques, energy-saving in-memory caching techniques, and other applications of external caching.

External caching for energy saving Early work on external caching was pioneered by Marsh *et. al* [MDK94], who proposed incorporating an ECD as part of the memory stack between the disk and memory. They proposed that all I/O traffic to the disk drive be cached/buffered in the ECD before continuing on its normal path. This technique, while having the potential to reduce the number of disk accesses, does not effectively utilize the ECD space by choosing carefully what to cache/buffer. Much more recently, Chen *et. al* [CJZ06] also propose to use the ECD to buffer writes, as well as prefetch and cache popular data. Their solution divides the ECD into zones dedicated for each optimization, as opposed to the unified buffer/cache technique of EXCES. Additionally, since they propose using read-ahead values at the VFS layer to anticipate future accesses, their solution does not have a clear presence in the I/O stack, with both block- and file- level concerns. Similarly, Bisson and Brandt proposed NVCache, an external caching system for energy savings [BBL06]. While the design of EXCES has some similarities to both NVCache and SmartSaver, EXCES differs in its implementation-oriented techniques to efficiently ensure data consistency under all conditions, its use of a novel page-rank algorithm tailored for increasing disk inactivity periods, and continuous and timely reconfiguration capability. More importantly, while all of the above studies evaluate their techniques on simulated models of disk operation and power consumption, we evaluate an actual implementation of EXCES with real-world benchmarks that re-

alistically demonstrate the extent of power-savings as well as impact to application performance.

In-memory caching for energy saving Weissel *et. al* [WBB02] and Papathanasiou *et. al* [PS04] propose to use cooperation/hints between the applications and the operating system. While Weissel *et al.* propose hints at the system call API for read/write operations, Papathanasiou propose using high-level hints about application I/O semantics such as sequentiality/randomness of access inside the operating system. Researchers have also looked at adaptive disk spin-down policies to complement in-memory caching techniques [PS04, HLS96, LKHA94]. We believe that all of the above can complement EXCES to further improve energy savings. Specifically, in this study, we compared EXCES against the open-source Laptop-mode tool [Sam04], and demonstrate that the Laptop-mode techniques complement EXCES well for some workloads to improve energy savings.

Other applications of external caching External caching has been used to improve I/O performance and reliability. Researchers have long argued for utilizing battery-backed caching (providing similar functionality as an ECD) for improving both reliability and performance [OD89]. Wang *et al.* [WRPH02] suggest using a Disk-ECD hybrid file system for improving application I/O performance by partitioning file system data into two portions, one stored on disk and the other on an ECD. More recently, the ReadyBoost [Mic] feature in the Windows Vista operating system utilizes an ECD if available to cache data. Since its primary objective is performance improvement, ReadyBoost directs small random read requests to the ECD and all other operations to the disk drive.

6.3 *fetch-before-update* Problem

Non-blocking writes, in concept, has existed for almost three decades for CPU cache lines. Observing that entire cache lines do not need to be fetched on a word write-miss in the cache, stalling the processor when doing so, additional registers that temporarily store these word updates to later be merged with the cache line was investigated. This idea was first introduced in the early eighties by Kroft [Kro81] to be used in CPU caches. Nowadays, this technique is widely used within modern CPU's like Intel Nehalem and Sun Niagara [LCBJ11].

We now discuss two simple approaches to mitigate the *fetch-before-update* problem. First is the simple approach of provisioning adequate DRAM to minimize out-of-core page writes. However, for both process memory and file system writes, the footprint of a workload over time is unpredictable and potentially unbounded. Moreover, technology trends do not support this as a viable solution; increasingly larger memory working sets are being supported with faster devices that serve as more cost effective and energy-efficient replacement for DRAM to store relatively cold data. Second, prefetching [SSS99] is an alternative approach that can reduce blocking by anticipating future memory accesses and prefetching necessary pages to eliminate page faults. Unfortunately, the use of prefetching is typically limited to sequential accesses to pages and it can incur both false positive and false negative page fetches that pollute memory. Non-blocking writes uses memory judiciously and only fetches those pages that are necessary for process execution. Ultimately however, prefetching and non-blocking writes are not exclusive and can be used in conjunction.

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. While the original implementation of AIO had a partial implementation for one file system (ext2) with support for file system page caching [BPPM03], the current state of Linux AIO is that it only works with the O_DIRECT flag, i.e., it does not work when file pages get cached in Linux [Sou] while FreeBSD and IRIX writes return only when the write I/O is queued implying that an out-of-core page write would still block for the page to be fetched first before it can be modified and written out to storage [BSD, *ni]. LAIO [ECCZ04] is a generalization of the basic AIO idea to make all system calls asynchronous; a library checkpoints execution state and relies on scheduler activation's to get notified about the completion of blocking I/O operations initiated inside the kernel. More 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 using a similar mechanism.

The scope of non-blocking writes in relation to the above proposals is different. It eliminates the blocking for memory writes to out-of-core pages in case of both supervised (system call based) as well as unsupervised (direct memory update in user-mode) access. However, unlike the above approaches, it is narrower in scope in the sense that does not eliminate blocking due to synchronous writing of resident memory pages to the backing store. A non-blocking write can be considered relatively lightweight since it does not use additional threads (often a limited resource in systems) to block on behalf of the running process nor does it need to checkpoint state thereby consuming lesser system resources. Finally, unlike these approaches

which require application modifications to use specific interfaces within libraries, non-blocking writes runs seamlessly in the OS transparent to applications.

Finally, there are some works that seem similar to non-blocking writes, but are actually quite different in their accomplished goal. First, speculative execution (or Speculator) as proposed by Nightingale *et al.* [NCF06] eliminates the blocking when synchronously writing cache 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 allows the process to make progress but does not allow it to externalize output before the write is committed to disk. Featherstitch [FMK+07] improves the performance of synchronous file system page updates by scheduling these page writes to disk more intelligently. 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 prior to committing the page to disk. Non-blocking writes thus presents a complementary improvement to the above body of work.

CHAPTER 7

CONCLUSIONS

In this thesis we designed and evaluated two complementary techniques to save energy in storage as well as a new technique to reduce the performance gap between commodity and energy efficient systems.

We began by presenting a new method to achieve fine-grained proportionality on multi-disk storage systems with reliability, workload shift adaptation, heterogeneity support, and low space overhead. SRCMap establishes the feasibility of such systems despite the few levels of energy consumption currently available in commodity disks. After implementing a prototype of the system, we found that SRCMap is able to save at least 35.5% of energy by spinning-down half of the disks on average when running 8 server-like workloads. We also found that the storage response time was not severely affected by having fewer disks spinning and that only 0.003% of the I/Os incurred in spin-up due to read misses. However, the synchronization I/Os can negatively affect the performance, implying that better scheduling techniques should be used to minimize this effect. Thanks to its dynamic adaptation to load, SRCMap requires nothing more than an initial manual tuning, freeing administrators of additional burden while still saving power. SRCMap has the potential for achieving even better results in data-centers where the scale of storage systems is typically much larger than what we used in our prototype.

We demonstrated that by caching and prefetching popular data in flash, single-disk systems were able to keep the disk spun-down longer. We implemented and tested our systems with several desktop-like workloads and found that EXCES was able to save between 2% and 14% of energy compared to the vanilla system, much lower savings that what previous works predicted. However, we found that the energy savings came at the cost of decreased performance due to the characteristics

of flash media. EXCES gives users the opportunity to increase the efficiency of their mobile devices by simply plugging in a common but energy efficient external device.

Finally, we presented non-blocking writes, a new method that eliminates the page *fetch-before-update* behavior by buffering writes to pages not available in memory and updating their content once they are loaded from disk. After implementing and evaluating non-blocking writes, we found a reduction of 50% in the average write latency of a file server benchmark. Moreover, when tested in single-threaded memory-intensive workloads, non-blocking writes was able to achieve a modest average reduction of 1% and a maximum of 8.7% in execution time. However, when exercised with multi-threaded benchmarks, non-blocking writes showed a degradation in the execution time compared to vanilla. We think this is due to sub-optimal scheduling of the CPU when using non-blocking writes with multi-threaded applications. In this situation, threads that only create patches occupy the CPU while preventing other threads from executing that are ready to run and have the potential to progress faster. To solve this problem, we proposed a new scheduler that gives higher priority to threads that have not created patches. This new scheduler guarantees that non-blocking writes will have the same execution time of vanilla in the worst case while using previously unutilized CPU cycles to increase performance.

We also evaluated non-blocking writes with different memory sizes and found that it is able to achieve higher benefits when memory is scarce. Due to its benefits in terms of execution time and latency reduction, non-blocking writes has the potential to close the performance gap between energy efficient systems and commodity systems. However, in the future, we need to find if energy efficient systems using non-blocking writes still have a positive balance in energy savings even though they have higher utilization of CPU.

To summarize, in this thesis, we demonstrated how caching techniques can be used to save energy both in multi-disk and single-disk systems. Moreover, we showed a solution to carefully control I/O activity that could further reduce the energy footprint of caching systems and improve the performance of low-memory energy-efficient systems. Overall, these solutions combined help advance the field of energy-efficient systems for both large-scale and personal computing. Observe that all the techniques presented herein may deliver better energy-efficiency and performance results when complemented with redesigned CPU schedulers, replacement algorithms, synchronization I/O schedulers, and specialized block allocation techniques optimized to use the underlying storage device according to their unique characteristics. Furthermore, future work should include a thorough evaluation of the benefits of combining the techniques we have presented in practical scenarios. Finally, as new storage technologies emerge, it would also be relevant to study the applicability and possible benefits of using these techniques in the newer devices.

In the next chapter, we discuss several directions for future research to follow up the work contained in this thesis.

CHAPTER 8

FUTURE WORK

The next challenge in energy efficient systems is memory. As energy efficient SSDs become more widely available, the energy share of storage in computer systems will start to decrease. In contrast, memory energy consumption continues to increase in response to applications demands for bigger DRAM. Hence, it is important to have a deeper understanding of how energy is consumed by DRAM and how to make it more efficient.

As discussed in previous chapters, DRAM energy consumption can be divided into idle and active power. The former accounts for the energy drawn by DRAM to constantly refresh its banks of memory. The later is used when data in memory is accessed.

Researchers have focused mostly on reducing the memory's active power by increasing locality or proposing new hardware changes. However, although DRAM idle power represents up to 30% of the memory's total energy used, operating system techniques to reduce memory idle power remain mostly unexplored. Idle power can only be decreased by reducing DRAM size. With less memory banks to refresh, the idle power is effectively reduced. Unfortunately, less memory also implies performance degradation. With less memory available, applications may incur in higher paging rates that would increase data access time. New energy efficient systems that reduce the memory idle power need to minimize this performance impact by: (a) making sure the working set of applications fits in the available DRAM and (b) decoupling the performance of backing store from the performance of memory accesses.

The first promising direction for future work is to hold the working set of applications by keeping in memory only the necessary data as opposed to full pages.

Commodity systems are limited to a minimum page size of 4KB even though applications may be using only a small portion of the page. With non-blocking writes we can effectively reduce the page size by keeping patches that correspond to the portions of a page that applications are actually using. Although this reduces memory requirements and idle power, it comes at the price of additional meta-data and possibly higher minor faults if used with virtual memory. Further research is necessary to understand the access pattern of applications and evaluate the feasibility of this approach.

A second research direction would be to increase the performance, and hence viability, of energy efficient systems that off-load low bandwidth memory accesses to flash devices. The idea is to explore how much the resident memory requirement can be reduced if we use flash cache systems (e.g. ZFS L2ARC) with non-blocking writes while maintaining the same performance as the non-flash cache configuration. New optimizations like a re-designed replacement algorithm that takes into account the difference in cost between read and write misses are also likely to be useful.

A final research direction involves exploring new techniques that leverage emerging technologies to dynamically power-off and power-on portions of DRAM as requested by the operating system. New techniques that continuously adapt the amount of DRAM while achieving a given applications' target performance become necessary. This approach also has the potential to increase the energy proportionality of memory. This idea can be combined with both of the previously proposed research directions to achieve even higher gains in energy efficiency.

Next, we present several additional follow up research directions to each of our proposed systems as well as further considerations when these systems are combined.

Energy Proportional Storage Our SRCMap work opens up new directions for further research. SRCMap can greatly benefit from better models of I/O workload intensity and correlation to reduce the performance impact of consolidated logical volumes. Further, improving the scheduling of synchronization I/Os can reduce the impact on foreground I/Os and increase the feasibility of SRCMap in commodity storage systems.

Energy-efficient Storage using Flash We believe that external caching systems offer a new direction for building energy saving storage systems. Improvements in ECD technology, especially in the performance dimension, can help accelerate the adoption of such systems. Our future work on EXCES will be directed towards the performance-sensitive server environment, where, in the absence of a display device, disk-drives would be the second highest power consuming component. Optimizations that address random write performance on the ECD will gain significant importance in such systems.

Controlling I/O Traffic in EXCES and SRCMap EXCES and SRCMap achieves energy efficiency by redirecting most of the I/Os to low-power caching devices. Their effectiveness is lost when applications start accessing data not contained in the cache. We can combine EXCES and SRCMap with non-blocking writes partially mitigate this problem and further improve their efficiency. If an application attempt to partially update a block, non-blocking writes could create patches of the updates and keep the primary device containing the full block undisturbed. Moreover, reads to any of the recently written data can be served as a read from patch case further increasing the idleness of the primary device. Since we found an increase in the CPU usage of non-blocking writes, we need to also evaluate the additional energy consumed and check that it does not surpass the benefits obtained.

Non-blocking writes and Multi-threading In previous chapters we discussed how non-blocking writes can harm multi-threaded applications if the CPU scheduling is not re-designed accordingly. Applications creating patches can deny the CPU to waiting threads that could potentially progress faster. This problem can be mitigated in two ways. First, we can implement a new CPU scheduling policy where threads are assigned priorities based on their rate of progress measured by instructions completed per unit of time. This will give threads with faster completion a bigger share of the CPU compared to threads creating mostly patches. Second, we can extend our implementation to include SMP. This will give ready-to-run threads more cores to execute rather than waiting until the time-slot of the offending thread expires.

Non-blocking writes and Page Replacement Non-blocking writes changes the relative importance of pages in memory fundamentally. There is less incentive to cache pages that are mostly written into in main memory. Even if the page were out-of-core, writes to these pages can be handled without synchronous demand paging or without demand paging at all. This observation can be built into page replacement algorithms which can now decrease the relative importance of pages that are frequently accessed but only (or mostly) written to.

Non-blocking Instructions Patches are a mechanism to define a data update in non-blocking writes when the data itself is available. There are certain machine instructions which generate a write fault but themselves do not contain the data to be written to. For instance, increment and decrement of a memory word (`INC/DEC` in x86 ISA) load the word from memory, increment it by 1, and then store the word back to memory. Creating a data patch from the instruction is not feasible. Creating an *instruction patch* on the other hand, is. The patch describes the instruction that

needs to be executed and the target memory address. Once the page is read in, the instruction patch can be applied by single-stepping the process with the instruction to generate the data update. This approach can also be applied to (`INC/DEC` with LOCK prefix in x86 ISA).

Non-blocking writes Additional Uses While file systems (including local and networked) and virtual memory systems are obvious use cases for non-blocking writes, there are other less obvious ones. The first use case is non-blocking writes at the hypervisor level. When memory is over-subscribed by virtual machines (VMs), the hypervisor starts an additional level of demand paging to maintain the illusion of available physical memory for the VMs [Wal02]. When the hypervisor takes a fault, the entire VCPU blocks and no process of the VM can run on the core, a situation worse than an OS taking a fault on account of a process. Non-blocking writes can reduce such VCPU blocking. The second use case is non-blocking writes is in post-copy live VM migration. Post-copy live VM migration is an optimization wherein the VM being migrated from a source to a target host starts running at the target without its state being migrated entirely to the target [HDG09]. This optimization allows us to migrate load away from an over-subscribed source host and thus mitigate performance issues earlier. During the initial execution at the target, many of the un-migrated state corresponds to pages that have been recently dirtied at the source and therefore not migrated in the iteration prior to the VM execution getting switched over to the target. As these pages continue to be written to at the target, the writes induce page fetches over the network since the source has more up-to-date versions of these pages.

BIBLIOGRAPHY

[ABO07] Jens Axboe, Alan D. Brunelle, and Others. blktrace user guide, February 2007.

[Adm11] U.S. Energy Information Administration. Monthly energy review, June 2011.

[Age07] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency, August 2007.

[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.

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

[BBL06] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. Nvcache: Increasing the effectiveness of disk spin-down algorithms with caching. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 422–432, September 2006.

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy proportional computing. In *IEEE Computer*, 2007.

[BH09] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, May 2009.

[BKB07] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *IEEE Conf. Integrated Network Management*, 2007.

[BKL+07] Len Brown, Konstantin A. Karasyov, Vladimir P. Lebedev, Alexey Y. Starikovskiy, and Randy P. Stanley. Linux laptop battery life: Measurement tools, techniques, and results, February 2007.

[Bla06] Blackburn, S. M. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of OOPSLA*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BPPM03] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous I/O Support in Linux 2.5. In *Proc. of the Ottawa Linux Symposium*, July 2003.

[Bro04] David Brownell. Linux usb "On-The-Go" (OTG) on OMAP H2, 2004.

[BSD] BSD. BSD System Calls Manual (aio_write). `http://www.unix.com/man-page/FreeBSD/2/aio_write/`.

[CG02] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *High Performance Networking and Computing Conference*, 2002.

[CJZ06] Feng Chen, Song Jiang, and Xiaodong Zhang. Smartsaver: Turning flash drive into a disk energy saver for mobile computers. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, pages 412–417, New York, NY, USA, 2006. ACM Press.

[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.

[Cor] HP Corporation. Hp storageworks san virtualization services platform: Overview & features. http://h18006.www1.hp.com/products/storage/software/sanvr/index.html.

[DKB95] Fred Douglis, P. Krishnan, and Brian N. Bershad. Adaptive disk spindown policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, Berkeley, CA, USA, 1995. USENIX Association.

[DMR+11] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. Memscale: Active low-power modes for main memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 225–238, New York, NY, USA, 2011. ACM.

[ECCZ04] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004.

[EM05] Jörn Engel and Robert Mertens. Logfs - finally a scalable flash file system, 2005.

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

[FMK+07] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. *Proc. of ACM SOSP*, pages 307–320, October 2007.

[GF07] Jim Gray and Bob Fitzgerald. Flash disk opportunity for server-applications. *Online,, http://research.microsoft.com/∼Gray/papers/FlashDiskPublic.doc*, January 2007.

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

[GLM+08] K. Greenan, D. Long, E. Miller, T. Schwarz, and J. Wylie. A spin-up saved is energy earned: Achieving power-efficient, erasure-coded storage. In *HotDep*, 2008.

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

[GSKF03] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. DPRM: Dynamic speed control for power man-

agement in server class disks. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*, June 2003.

[GSS09] S. Gurumurthi, M. R. Stan, and S. Sankar. Using intradisk parallelism to build energy-efficient storage systems. In *IEEE MICRO Top Picks*, 2009.

[GWBB07] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman. Optimizing power consumption in large scale storage systems. In *HotOS*, 2007.

[Hag87] Robert Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *Proc. of the ACM Symposium on Operating systems principles*, November 1987.

[HDG09] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43:14–26, July 2009.

[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 *Proc. of the ACM Symposium on Operating Systems Principles*, October 2011.

[HLS96] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

[HLSS00] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, page 297, 2000.

[HP06] HP. Control power and cooling for data center efficiency - hp thermal logic technology. an hp bladesystem innovation primer. http://h71028.www7.hp.com/ERC/downloads/4AA0-5820ENW.pdf, 2006.

[IBM] IBM Corporation. Ibm system storage san volume controller. http://www-03.ibm.com/systems/storage/software/virtualization/svc/.

[IDC06] IDC. Virtualization across the enterprise, Nov 2006.

[Kat97] Jeffrey Katcher. PostMark: A New File System Benchmark. *Technical Report TR3022. Network Appliance Inc.*, October 1997.

[KAU12] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proc. of USENIX File and Storage Technologies (to appear)*, February 2012.

[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, pages 103–112, New York, NY, USA, 2006. ACM.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, pages 81–88, 1981.

[KS07] Patricia Kim and Mike Suk. Ramp load/unload technology in hard disk drives. *Hitachi Global Storage Technologies White Paper*, 2007.

[Lan09] Klaus-Dieter Lange. Identifying shades of green: the specpower benchmarks. *computer*, 42:95–97, 2009.

[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.

[LKHA94] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the USENIX Winter Conference*, 1994.

[LLN08] H. Lee, K. Lee, and S. Noh. Augmenting raid with an ssd for energy relief. In *HotPower*, 2008.

[LPGM08] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system-workloads. In *Usenix ATC*, 2008.

[LVW07] Lanyue Lu, Peter Varman, and Jun Wang. Diskgroup: Energy efficient disk layout for raid1 systems. *Networking, Architecture, and Storage, International Conference on*, 0:233–242, 2007.

[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.

[MDK94] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, 1994.

[Mic] Microsoft Corporation. Windows Readyboost. Online, http://www.microsoft.com/windows/products/windowsvista/ features/details/readyboost.mspx.

[MV04] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *PACS*, pages 165–180, 2004.

[NC] William D. Norcott and Don Capps. The Iozone File System Benchmark. http://www.iozone.org/.

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

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 17:1–17:15, Berkeley, CA, USA, 2008. USENIX Association.

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

[*ni] *nix Documentation Project. IRIX Man Pages (aio_write). `http://nixdoc.net/man-pages/IRIX/man3/aio_write.3.html`.

[NVCF06] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. *Proc. 7th USENIX OSDI*, Nov 2006.

[OD89] John K. Ousterhout and Fred Douglis. Beating the i/o bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, 1989.

[PADAD05] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[PB04] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 68–78, New York, NY, USA, 2004. ACM.

[PBD06] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *SIGMETRICS*, 2006.

[PS04] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[RLLR07] Alma Riska, James Larkby-Lahet, and Erik Riedel. Evaluating block-level optimization through the io path. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 19:1–19:14, Berkeley, CA, USA, 2007. USENIX Association.

[Sam04] Bart Samwel. Kernel korner: extending battery life with laptop mode. *Linux J.*, 2004(125):10, 2004.

[SCN+10] Kshitij Sudan, Niladrish Chatterjee, David W. Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing dram efficiency with locality-aware data placement. In James C. Hoe and Vikram S. Adve, editors, *ASPLOS*, pages 219–230. ACM, 2010.

[SGMV08] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable disk-based archival storage. In *Usenix FAST*, 2008.

[Sou] Sourceforge. Kernel Asynchronous I/O (AIO) Support for Linux. `http://lse.sourceforge.net/io/aio.html`.

[SS10a] Mohit Saxena and Michael M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 14–14, Berkeley, CA, USA, 2010. USENIX Association.

[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, Berkeley, CA, USA, 2010. USENIX Association.

[SSS99] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why does file system prefetching work? In *Proc. of USENIX ATC*, 1999.

[Sta] Standard Performance Evaluation Corporation (SPEC). SPEC Benchmarks. `http://www.spec.org/benchmarks.html`.

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

[TWM+08] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering Energy Proportionality with Non Energy-Proportional Systems – Optimizing the Ensemble. In *HotPower '08: Workshop on Power Aware Computing and Systems*. ACM, December 2008.

[UGB+08] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami. Exces: External caching in energy saving storage systems. In *HPCA*, 2008.

[UKRV11] Luis Useche, Ricardo Koller, Raju Rangaswami, and Akshat Verma. Truly non-blocking writes. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, pages 8–8. USENIX Association, 2011.

[VAN08] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and migration cost aware application placement in virtualized systems. In *Middleware*, 2008.

[VDN+09] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Usenix ATC*, 2009.

[VKUR10] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[VMw] VMware. The Role of Memory in VMware ESX Server 3. `http://www.vmware.com/pdf/esx3_memory.pdf`.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[Wat09] Wattsup Corporation. Watts up? PRO Meter. https://www.wattsupmeters.com/secure/products.php?pn=0, 2009.

[WBB02] Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative i/o - a novel i/o semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[WOQ+07] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter L. Reiher, and Geoffrey H. Kuenning. Paraid: A gear-shifting power-aware raid. *TOS*, 3(3), 2007.

[WR10] Xiaojian Wu and A. L. Narasimha Reddy. Exploiting Concurrency to Improve Latency and Throughput in a Hybrid Storage System. In *Proc. of IEEE MASCOTS*, September 2010.

[WRPH02] An-I A. Wang, Peter Reiher, Gerald J. Popek, and Geoffrey H.Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[WYZ08] J. Wang, X. Yao, and H. Zhu. Exploiting in-memory and on-disk redundancy to conserve energy in storage systems. In *IEEE Tran. on Computers*, 2008.

[ZCT+05] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP*, 2005.

VITA

Luis Useche

July 27th, 1983	Born, Caracas, Venezuela
2006	B.A., Computer Science Engeneering Simón Bolívar University Caracas, Venezuela

PUBLICATIONS AND PRESENTATIONS

Luis Useche, Ricardo Koller, Raju Rangaswami, Akshat Verma, (2011). *Truly Non-blocking Writes*. Proceedings of USENIX 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage).

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

Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami and, Vagelis Hristidis, (2009). *BORG: Block-reORGanization for Self-optimizing Storage Systems*. Proceedings of File and Storage Technologies (FAST).

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

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