7-13-2012

Improving Storage with Stackable Extensions

Jorge Guerra
Florida International University, jguerra@cs.fiu.edu

DOI: 10.25148/etd.FI12080630

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

Improving Storage with Stackable Extensions

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jorge Guerra

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Jorge Guerra, and entitled Improving Storage with Stackable Extensions, 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.

Kaushik Dutta

Himabindu Pucha

Jinpeng Wei

Ming Zhao

Raju Rangaswami, Major Professor

Date of Defense: July 13, 2012

The dissertation of Jorge Guerra is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

DEDICATION

I like to dedicate this thesis to my parents. Your love and guidance has made me believe in myself.

ACKNOWLEDGMENTS

First and foremost I would like to thank Raju, my advisor, working with you has been an extraordinary experience, and many times you have gone the extra mile and supported me both academically and in personal life. I am very thankful for everything you thought me, and will remmeber it for my entire life. This work could simply have not been possible without the endless and unconditional love and support of Karen, my love, thank you for always standing by my side and helping me back up when I have fell.

I would like to give special thanks to my family, in particular my parents José Guerra and Xiomara Delgado, and my brother Rodrigo Guerra. You have always provided me love, help and support everytime I was in need, and I consider myself very privileged to have all you by my side.

I would like to thank all the members of my thesis committee: Kaushik Dutta, Jason Liu, Himbabindu Pucha, Jinpeng Wei and Ming Zhao. Thank you for serving in my committee and also for your comments, suggestions and feedback all which have helped strengthen this dissertation.

I would also like to thank the members of the Systems Research Laboratory at FIU: Medha Bhadkamkar, Daniel Campello, Carlos Crespo, Fernando Farfán, Ricardo Koller, Sajib Kundu, Leonardo Mármol and Luis Useche. It has been an absolute delight to interact with all of you.

Finally, I would like to thank IBM for providing support for the final two years of my PhD and for giving me the opportunity to spend three very insightful summer at the Almaden research center working with Wend Belluomini, Jody Glider and Bindu Pucha an amazing team from which I learned and grew greatly. I am also very thankful to NSF for providing the necessary funding for the other half of my studies, as well as most of equipment used for this work.

ABSTRACT OF THE DISSERTATION

IMPROVING STORAGE WITH STACKABLE EXTENSIONS

by

Jorge Guerra

Florida International University, 2012

Miami, Florida

Professor Raju Rangaswami, Major Professor

Storage is a central part of computing. Driven by exponentially increasing content generation rate and a widening performance gap between memory and secondary storage, researchers are in the perennial quest to push for further innovation. This has resulted in novel ways to "squeeze" more capacity and performance out of current and emerging storage technology. Adding intelligence and leveraging new types of storage devices has opened the door to a whole new class of optimizations to save cost, improve performance, and reduce energy consumption.

In this dissertation, we first develop, analyze, and evaluate three storage extensions. Our first extension tracks application access patterns and writes data in the way individual applications most commonly access it to benefit from the sequential throughput of disks. Our second extension uses a lower power flash device as a cache to save energy and turn off the disk during idle periods. Our third extension is designed to leverage the characteristics of both disks and solid state devices by placing data in the most appropriate device to improve performance and save power.

In developing these systems, we learned that extending the storage stack is a complex process. Implementing new ideas incurs a prolonged and cumbersome development process and requires developers to have advanced knowledge of the entire system to ensure that extensions accomplish their goal without compromising data recoverability. Futhermore, storage administrators are often reluctant to deploy

specific storage extensions without understanding how they interact with other extensions and if the extension ultimately achieves the intended goal. We address these challenges by using a combination of approaches. First, we simplify the storage extension development process with system-level infrastructure that implements core functionality commonly needed for storage extension development. Second, we develop a formal theory to assist administrators deploy storage extensions while guaranteeing that the given high level goals are satisfied. There are, however, some cases for which our theory is inconclusive. For such scenarios we present an experimental methodology that allows administrators to pick an extension that performs best for a given workload. Our evaluation demostrates the benefits of both the infrastructure and the formal theory.

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 7

3. Related work 9
3.1 Extensions that Improve the Storage System 9
3.2 Storage Infrastructures 11
3.3 Persistent Memory 13

4. Block Layer Extensions 17
4.1 A Block Layer Solution. 18
4.2 Optimizing Data Layout for Performance 20
4.2.1 BORG System Architecture 23
4.2.2 BORG Evaluation Results 25
4.2.3 BORG Summary 25
4.3 Lowering Power Consumption with External Caching 26
4.3.1 EXCES: System Architecture 28
4.3.2 EXCES Summary 29
4.4 Improving Performance with Dynamic Multi-tier Systems 30
4.4.1 Multi-Tiering Design Choices 32
4.4.2 Design Overview 35
4.4.3 Configuration Adviser 40
4.4.4 Dynamic Tier Manager 41
4.4.5 EDT Summary 45
4.5 Lessons Learned from Implementing Storage Extensions 46
4.5.1 Correctly Migrating Data 46
4.5.2 Indirection Implementation Issues 47
4.5.3 Maintaining a Persistent Indirection Map 48
4.5.4 Programming Inside the OS Kernel 48
4.6 Summary 50

5. Building and Stacking Storage Extensions 51
5.1 Infrastructure Requirements 52
5.1.1 Requirements of the Developer 52
5.1.2 Requirements of the Administrator 53
5.2 ABLE Architecture and Design 54
5.2.1 ABLE Architecture and Design 54

5.2.2 The ABLE Primitives . . . 56
5.2.3 ABLE and File Systems . . . 58
5.3 A Theory of Block Layer Extensions . . . 59
5.3.1 An Illustrative Example . . . 59
5.3.2 Concepts and Definitions . . . 60
5.3.3 Generalized Extension Stacking . . . 64
5.3.4 On Effectiveness and Theory Limitations . . . 66
5.3.5 An Illustration of Stacking Rules . . . 67
5.4 Putting Theory into Practice . . . 70
5.4.1 The Administration Scenario . . . 70
5.4.2 Configuring Extensions and System Goals . . . 71
5.4.3 Identifying the Stacking Order . . . 72
5.4.4 Automatic Extension Stack Composition . . . 74
5.5 Evaluating the ABLE Infrastructure . . . 75
5.5.1 Complexity Metrics . . . 75
5.5.2 ABLE Implementation Details . . . 76
5.5.3 Simple RAID-1 Illustration . . . 76
5.5.4 Quantitative Complexity Evaluation . . . 78
5.6 ABLE Runtime Overheads . . . 80
5.7 Summary . . . 81

6. Comparing Extensions . . . 82
6.1 Problem Background . . . 82
6.2 Understanding Workload Characteristics . . . 85
6.2.1 Storage I/O Distribution . . . 87
6.2.2 I/O Distribution Through Time . . . 88
6.3 Overview of Two Approaches . . . 89
6.3.1 Caching . . . 90
6.3.2 Multi-Tiering . . . 98
6.4 Caching vs. Multi-tiering . . . 102
6.4.1 SSD Effectiveness . . . 103
6.4.2 Workload Adaptability . . . 105
6.4.3 Leveraging Device Heterogeneity . . . 108
6.4.4 Metadata Overhead . . . 109
6.4.5 Reliability . . . 111
6.5 Discussion and Summary . . . 112

7. Making Extension Data Persistent . . . 114
7.1 Persistence of Memory . . . 115
7.1.1 Design Goals . . . 115
7.2 SoftPM Overview . . . 116
7.2.1 System Architecture . . . 118
7.3 LIMA Design . . . 118

7.3.1 Container Manager . . . 119
7.3.2 Discovery and Memory Allocation . . . 120
7.3.3 Write Handler . . . 121
7.3.4 Flusher . . . 122
7.4 SID Design . . . 122
7.4.1 SID Basics . . . 123
7.4.2 Device-specific optimizations . . . 125
7.5 Pointer Detection . . . 126
7.6 Evaluation . . . 127
7.6.1 Workloads . . . 128
7.6.2 Correctness Evaluation . . . 129
7.6.3 Case Studies . . . 130
7.6.4 Microbenchmarks . . . 135
7.7 Discussion . . . 138
7.8 Summary . . . 140

8. Future Work . . . 142

9. Conclusions . . . 145

BIBLIOGRAPHY . . . 147

Appendices . . . 163

Vita . . . 176

LIST OF TABLES

TABLE PAGE

4.1 Summary statistics of week-long traces obtained from four different systems. . . . 21

4.2 Various laptop configurations used in profiling experiments. . . . 27

5.1 ABLE primitives (not a complete list). . . . 56

5.2 I/O domain configurations for $\mathcal{DD}$ and $\mathcal{IO}$. . . . 71

5.3 Development complexity statistics for extensions. . . . 79

6.1 Access times throughout the memory hierarchy. Numbers reported are approximations taken from the device specification sheets. . . . 84

6.2 Summary of Traces Studied 85

6.3 Amount of the metadata for the multi-tier. Assuming 20 byte per extent metadata. . . . 101

6.4 Device Cost Comparison from 2010 to 2012. *SSD'10* is a 120GB Intel x25 MLC, *SSD'12* is a 120GB Intel 320 MLC, *SAS* is a 450GB 3.5in 15K RPM SAS, and *SATA* is a 1TB 3.5in 7.2K RPM SATA 105

7.1 The SoftPM application programmer interface. . . . 117

7.2 Persistent structures used in application and systems software. Arrays are multidimensional in some cases. *O* indicates other (e.g., graphs) and/or hybrid structures. In some cases, we assume an implementation based on design descriptions. This summary is created based on descriptions within respective articles and/or direct communication with the developers of these systems. . . . 130

7.3 Lines of code to make structures (or applications) persistent and recover them from disk. We used TPL for Array, Linked List, RB Tree, and Hash Table; SQLite and memcachedb implement custom persistence. . . . 131

A.1 Characteristics of devices used in the testbed. . . . 165

A.2 Configuration for synthetic workload. The number of disks per tier is specified as (SSD, SAS, SATA). The average response time is obtained from running the configuration with 100 BSUs 167

A.3 Configuration for *MSR-combined*. Configurations achieving equal performance depict improvement in cost and peak power. Configurations at equal cost are created for experimental ease. Number of disks in each tier specified as (SSD, SAS, SATA). . . . 168

A.4 Sub-workloads derived from MSR. . . . 171

A.5 Configuration for MSR sub-workloads. Number of disks in each tier specified as (SSD, SAS, SATA). 172

LIST OF FIGURES

FIGURE PAGE

4.1 Storage Layers and information available. 18

4.2 Rank-frequency, heatmap, and working-set plots for week-long traces from four different systems. The heatmaps (middle row) depict frequency of accesses in various regions of the disk, each cell representing a region. Six normalized, exponentially-increasing heat levels are used in each heatmap where darker cells represent higher frequency of accesses to the region. Disk areas are mapped to cells in row-major order. 22

4.3 BORG System Architecture. 23

4.4 Power consumption profiles of various ECD types and interfaces. 28

4.5 EXCES system architecture. 28

4.6 EDT system architecture. 35

4.7 Lowest cost tier for extents with different characteristics. 39

4.8 Possible cases when migrating data 46

4.9 Alignment problem example 47

5.1 ABLE architecture. Arrows depict control flow. 54

5.2 Stacking options for extensions $\mathcal{IS}$ and $\mathcal{EX}$. Lines represent I/O request flow, thicker lines indicate greater I/O flow, gray and dashed lines indicate requests created by $\mathcal{IS}$ and $\mathcal{EX}$ respectively. D is the disk drive and F is a flash device. 60

5.3 Extension Stacking Rules Grid. These rules assume there are only two extensions in the stack, A and B, with A stacked over B. x is a block (represented by its address) within some data volume managed by A and B. I_K and O_K represent the input and output domains of an extension K respectively. $f_K(x)$ is the resultant request after x is processed by extension K. 64

5.4 Domains definitions for sample extensions. 68

5.5 Sample stack configurations illustrating the stacking rules grid elements depicted in Figure 5.3. 69

5.6 I/O handling with alternative stacking of extensions $\mathcal{IO}$ and $\mathcal{DD}$. Arrows represent I/O requests. d_1 and d_2^r are two writes with identical contents originally addressed to the local and remote volumes respectively. 73

5.7 ABLE component statistics. 76

5.8 Function pseudo-code and LOC comparison for Simple RAID-1 extension. 77

5.9 Normalized extension processing times. Prefixes V and A refer to the vanilla and ABLE versions respectively. Results are averaged over five runs. 80

6.1 I/O distribution through the data. The bars depict the % of I/Os the issued to the top X% of data accessed. The solid curve depicts the same information as a cumulative distribution. 87

6.2 I/O operations per second across time for the MSR and FIU workloads. 88

6.3 System architecture with (a) shared and (b) local SSDs 89

6.4 Architecture diagrams of the proposed caching and tiering schemes incorporating SSDs. 90

6.5 Hit rate of LRU and MQ caching algorithms for the 7 day period, using 16 KB chunks and a 100 GB cache. 92

6.6 Hit ratio for different cache sizes. 93

6.7 Obtaining the miss hit rate curve from an access sequence. (a) Access sequence and actual hit ratio for caches of different sizes, (b) Reuse distance of the sequence, (c) Hit ratio estimate calculated using the reuse distance, dotted line indicates actual hit ratio. 94

6.8 Hit ratios for different cache line sizes for the MSR and FIU workloads. Lines depict the cache line size. 95

6.9 Distribution of inter I/O distance for the MSR and FIU workloads. . . . 97

6.10 I/O Service times for the tiering system varying the algorithm during hours 6-8 of the MSR workload, using 16MB chunk and 30min epochs. 99

6.11 Cumulative distribution of response time for the multi-tier system during hours 120-122 of the MSR workload. 101

6.12 I/O Service times for the 5^{th}, 25^{th}, 50^{th} and 95^{th} percentiles for the tiering system varying the algorithm during hours 6-10 of the MSR workload, using 16MB chunk and varying the epochs length. 103

6.13 SSD hit ratio for a replay of hours 120 to 122 of the MSR traces using a 200 GB SSD. 104

6.14 Distribution of response time for a (a) workload that exhibits a chance in its working set, and (b) one the honors long term trends. 107

6.15 Average service time per device for the Caching and Tiering systems for the 120-122 hour period. 109

6.16 Metadata overheads for the two techniques. The calculations assume the SSD is 10% the total size of the data; 3x ghost buffer size, with 8 byte metadata per chunk for caching. For tiering we assume 20 bytes of metadata per chunk. (a) Assumes that 50% chunks are never accessed, (b) Assumes chunk size of 64 KB. 110

6.17 Amount of 1 MB data movements between SSD and HDD incurred by caching and multi-tiered systems during the reply of hours 120-122. . 111

7.1 Implementing a persistent list. 116

7.2 The SoftPM architecture. 117

7.3 Container virtual address spaces in relation to process virtual address space and LIMA/SID volumes. The container virtual address space is chunked, containing a fixed number of pages (three in this case). 119

7.4 Container Discovery. Grey boxes indicate freed memory. 120

7.5 Performance of individual data structure operations. The bars represent the execution time of the SoftPM version relative to a version that uses the TPL serialization library for persistence. We used fixed size arrays which do not support add or remove operations. 131

7.6 Performance of memcachedb using different persistent back-ends. The workload randomly adds, queries, and deletes 512 byte elements with 16 byte keys. The dashed line represents a memory only solution. 133

7.7 SQLite transactions per second comparison when using SoftPM and the native file persistence. 133

7.8 Breakdown of time spent in the SQLite benchmark for 100K rows. . . . 134

7.9 Contrasting application execution times for MPI matrix multiplication using 9 processes. 134

7.10 Impact of locality on incremental persistence. Two different sets of experiments are performed: *(m)* where only the contents of the nodes are modified, and *(a/r)* where nodes are added and removed from the list. In both cases the size of the list is always 500MB. 136

7.11 Impact of chunk size on persistence point time. A list of size 500MB is made persistent and individual lines depict for a specific fraction of the list modified. 136

7.12 Time to create persistence points for multiple parallel processes. Every process persists a list of size (1GB/number-of-processes). 137

7.13 Time to persist a linked list and LIMA metadata size, varying percentage of pointers in data. The total size is fixed at 500MB and node sizes are varied accordingly. 138

8.1 Different storage architectures 142

A.1 Storage subsystem platform for evaluating EDT-CA and EDT-DTM. . . 163

A.2 I/O rate and power consumption (left) and response time distribution (right) for *MSR-combined*. 168

A.3 Contrasting extent migrations for EDT and IDT. The two upper lines denote extent placement for the different algorithms. Black is SSD tier, dark grey SAS and light grey SATA. 171

A.4 Replaying 6 hours of the MSR sub-workloads. First column is *server*, second *data*, and third *srccntl*. 173

A.5 Extent distribution and CDF for the adversarial workload. 174

CHAPTER 1

INTRODUCTION

Every day we create an increasing amount of data including emails, photos, and videos, all of which must be accessible in a timely and reliable manner. This data can be stored in our personal computers or in data centers around the world. On account of the growing data requirements, storage is rapidly becoming an important component in data center IT equipment. A recent survey by Gartner, Inc. [Gar10] reveals that data growth is the biggest challenge for large enterprises. However, storage not only needs to scale in size, but also in performance, reliability, and power efficiency, among others; all these challenges must be met while minimizing deployment and administration efforts. Enterprises are actively taking measures to mitigate the growing data problem.

Recent years have witnessed substantial innovation in storage systems extensions that provide critical improvements in meeting some of these storage system goals [FMK+07, GPK+07, LCSZ04, MAC+08, Nar08, NDT+08, SWS05, ZLP08]. However, many challenges remain unmet. In this dissertation, we illustrate the opportunities and challenges present in developing storage extensions with first-hand experiences from building three extensions of practical importance. First, in BORG [BGU+09] we present a self-optimizing storage extension that performs *automatic block reorganization* based on the observed I/O workload. BORG manages a small, dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition to significantly reduce seek and rotational delays. This work shows improvements of up to 60% reduction in application execution time. Our second extension, EXCES [UGB+08] presents the design and implementation of an extension that employs prefetching, caching, and buffering of disk data for reducing disk activity. Our evaluation showed overall system en-

ergy savings to lie in the modest 2-14% range, depending on the workload. Finally, we designed and implemented *Extent based Dynamic Tiering* (EDT) [GPG+11], a system that dynamically optimizes data placement among storage tiers to improve performance and save power. The evaluation revealed that multi-tier systems using EDT have a device mix that saves between 5% to 45% in cost, consumes up to 54% less peak power at a better or comparable performance compared to a homogeneous SAS storage system.

During our experience building these extensions, we found the development to be a cumbersome process which can take months or even years. Adding to this complexity is the fact that in most cases it requires modifying an operating system kernel, an inherently complex task. We believe that creating an infrastructure that addresses these issues will ease the development of storage extensions and provides a crucial step towards the rapid development and adoption of novel ideas. Such infrastructure provides functionality commonly used in block layer extension development, including mechanisms to safely move data and interact with the kernel I/O path. We also found that extensions typically create and manage internal metadata that must be persistent for ensuring correctness of recovery from failures. For instance, EXCES maintains a mapping indicating which blocks are in the cache. In most cases this metadata must be made persistent and durable to protect against crashes and other system failures. Achieving this durability has been no easy task [RO91, PADAD05, PBA+05]. We create a system that automatically discovers data that must be made persistent. Developers only need to specify which data structures to persist, and our system automates storing and restoring the data. We demostrate that by using this infrastructure developers are able to write less amount of code allowing them to focus on effectively implementing their core solution.

Creating new storage extensions only solves part of the problem. System administrators then need to deploy these extensions. In many cases they need to choose from various extensions that accomplish the same goal, for instance SSD-based caching and multi-tiering (EDT). Both extensions leverage the benefits from different types of storage devices, each employing a different set of techniques to accomplish its goal. Thus, determining which extension is a better choice for a given scenario is not straight forward. We present an empirical study that illustrates how administrators can select the extension that better enables meeting the high level goal, in this case performance.

Finally, once the administrator has chosen which extensions to deploy, the extensions still need to be added to the storage stack and interact with other extensions which are already deployed. Understanding such interactions is not a trivial task. For example, if we want to deploy an extension that caches popular blocks in memory and a RAID1 extension, putting the caching extension on top may reduce reliability, but placing it below the RAID1 will lead to a cache that has duplicates for every block. In most cases developers have to be aware of how the extension would be deployed while administrators need to be be aware of how the extension works to determine the appropriate stacking order. This slows adoption of novel ideas in storage systems. We believe that extension stack composition should be driven by the ability to satisfy service goals given to the system. Thus, we develop a formal theory that allows administrators to quickly determine a stacking order that performs best according to a specified high level goal, such as improving performance or saving power. We demostrate this theory to be useful in a set of real life scenarios when administrators want to improve their storage systems, which gives them the ability to know the influence of the extensions to be deployed on the storage system.

The rest of this thesis is structured as follows. In chapter 2 we identify the thesis statement and discuss its significance. We study the related work in chapter 3. Later we present three novel storage system extensions and comment on the lessons learned while developing these in chapter 4. In chapter 5 we propose an infrastructure to develop storage system extensions based on a formal theory that helps administrators deploy such extensions. We complement this theory with an empirical methodology to determine which extension is more appropriate for a workload in chapter 6. Next, we present a persistent memory system that will further facilitate storage extension development in chapter 7. In chapter 8 we discuss possible future work directions. We end with some concluding remarks in chapter 9.

CHAPTER 2

PROBLEM STATEMENT

This section introduces the core research problem addressed in this proposal. We start with a clear statement of the thesis, elaborate on its significance, introduce specific challenges that we address, and our unique contributions.

2.1 Thesis Statement

In this dissertation, we improve the state-of-the-art in block layer storage by:

(i) designing and implementing several self-managed block layer extensions that utilize existing and emerging storage technology in novel ways,

(ii) creating an infrastructure to simplify the development of block layer extensions, and

(iii) developing a theory that simplifies the deployment and management of the extensions in production systems.

2.2 Thesis Statement Description

The past 20 years have seen a significant amount of innovation in storage systems both from industry and academia. Some of these innovations became very successful and are now ubiquitous, e.g. RAID [PGK88]. But there is still substantial work to done. Demand for storage is increasing exponentially and newer types of devices are being added to the storage stack. These shifts bring additional possibilities for improvement.

The first contribution of this thesis are the lessons learned from designing and implementing several self-managed block layer extensions that utilize existing and emerging storage technology in novel ways. One area of interest is the widening

performance gap between main memory and disks. This gap is making disk I/O the bottleneck for a wide range of workloads. However, a significant source of the problem is accessing disks in a non-sequential manner; doing so can degrade I/O performance by up-to a factor of 100x. We built and evaluated a system that dynamically adapts to the observed workload by tracking per application requests and optimizing these for sequential access. Another area of particular interest is how to effectively leverage the characteristics of new types of storage devices. The second extension studied considers reducing power consumption by adding inexpensive flash devices as cache trying to capture as much I/O as possible to the flash device and spindown the disks. Third, we build a dynamic system that incorporates three storage tiers (SATA disks drives, SAS/FC disks drives, and solid state drives) and moves data between devices depending on access characteristics to improve performance and/or save energy. Finally, we build a modular caching infrastructure that allows us to evaluate the performance implications of the different design decisions one taken when building a storage caching system.

Based on our lessons developing those extensions, the second contribution of this thesis is creating an infrastructure to simplify the development of block layer extensions. In our studies, we have noticed that although many storage extensions have been developed through the years, their mainstream adoption has been slow in most cases. We believe this is due to several reasons. Developers must create extensions for storage systems which are very complex and find themselves spending significantly more time than originally scheduled to bring a simple idea to reality. The implementation and testing processes are also cumbersome because of the complexity of the storage stack. What further complicates the process is the fact that possible interactions with other systems have to be considered, always keeping in mind the unforgiving requirement of data consistency. We build a block stor-

age development infrastructure which simplifies the extension building process by providing commonly used functionality and support for persistent data structures.

The third contribution of this thesis is developing a theory that simplifies the deployment and management of the extensions in production systems. Administrators are responsible for maintaining data consistent and readily accessible. They often fear adding more functionality to their systems without a definitive understanding of how it is going to interact with their current system. This is limiting widespread deployment of novel storage extensions. We believe that if administrator are provided with a concrete method to reason about the effect of incorporating a particular extension to the current stack and its overall effect on the system we would evidence more innovations in storage systems being adopted. We create a extension stacking theory to help them decide how to incorporate extensions to a given stack in order to accomplish established high level goals.

2.3 Thesis Significance

The demand for storage is growing at an extraordinary rate. With this growth users are demanding storage systems that are faster, more efficient, and have higher capacity. Storage system developers bear the responsibility for creating systems that accommodate for all the users needs and bring innovation to users in a timely fashion insuring at all times data consistency. And finally, administrators require storage systems that are reliable and easy to manage.

This dissertation presents solutions that simplify the tasks of both developers and administrator, and by extension provide a better experience for the user. First, we present a series of novel storage system extensions that improve the system's performance; this for the benefit of the users. For developers, we build an infrastructure the simplifies the storage extension development process thereby potentially

accelerating the pace at which new innovations are added to the storage system. We formalize reasoning to help administrators deploy these extensions to obtain a better performing system and in accordance with their goals.

CHAPTER 3

RELATED WORK

In this section we examine the existing work for the systems we present in the later sections. Previous work for this proposal covers a wide range, from storage extensions with similar functionality to kernel development infrastructures to persistent memory systems. This section is structured as follows. First, we examine the literature for work related to the extensions that we have built and identify what differentiates us. Next, we look into previous work regarding in-kernel development libraries. We conclude by reviewing work for persistent memory systems.

3.1 Extensions that Improve the Storage System

There is an extensive amount of previous work regarding developing extensions for systems. On the optimizations for data layout we find various approaches. Early work [Won80, SGM91] argued for placing the frequently accessed data in the center of the disk. More recently, other researchers have focus on optimizing application startup [Int98, Mic06, hfs04]. BORG is a generic solution in comparison to the above approaches, since it creates a block reorganization mechanism that can adapt to an arbitrary workload. Perhaps the closest work to BORG is FS2 [HHS05], which proposes replication of frequently accessed blocks based on disk access patterns in file system free space. This strategy, unfortunately, also restricts the degree of seek and rotational-delay optimization due to the distribution of free space. Since FS2 may create multiple copies of a block simultaneously, staleness, and consequently, space and I/O bandwidth wastage, become important concerns; BORG maintains at most one extra copy of each block and its strength is in being a non-intrusive, storage-stack friendly, and file system independent (portable) solution.

Regarding, previous work on external caching, Marsh *et. al* [MDK94] was the first who proposed incorporating an external device as part of the memory stack between the disk and memory. Chen *et. al* [CJZ06] proposed to use the cache to buffer writes, as well as prefetch and cache popular data. Others have solutions that address the issue of reducing disk energy consumption include adaptive disk spin down policies [DKB95, HLS96, LKHA94], exploiting multi-speed drives [GSKF00, PB04, ZCT+05, ZDD+04], using data migration across drives [CPB03, CG02], and energy-aware prefetching and caching techniques [PS04, WBB02]. 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 realistically demonstrate the extent of power-savings as well as impact to application performance. More recently we have witnessed a substantial amount of research has focused on specializing caching algorithms for known disk access patterns [MM03, JS94, OOWZ93, ZCL04, WW02, JZ02].

Driven by advances of solid state memories multiple storage vendor have began to incorporate SSDs as caches for disks. Some of the solutions with caching include NetApp's FlashCache [Pet09b], Oracle (via Sun) ZFS storage appliances 7000 series [Ora10], and Nimble storage CS-series [Nim10]. The design differences between these systems can be significant. For instance, while FlashCache and Nimble favor using flash-based storage as a read-only cache, Oracle's Unified Storage solution uses SSDs to cache both reads (L2ARC) and writes (ZIL) [Gre08]. However, it is up for debate if this is the most effective way to incorporate SSDs into the storage for enterprise workloads [Mar10, Owe10] since other researchers and storage companies are pushing for multi-tiering based solutions.

Among the commercially available multi-tier systems we find 3PAR [Pet10], IBM's EasyTier [Tan10], EMC's FAST [Lal09], and Compellent [Pet09a] systems

to incorporate SSDs in storage tiering solutions. Nevertheless, since technical details of these approaches are not published, the extent based dynamic tiering (EDT) that we propose is the first to provide insight into design choices and components, detailed evaluation across workloads, and analysis of benefits and challenges in building SSD-based multi-tier systems. Moreover, the publicly available documents of these products indicate that although they achieve cost savings and performance improvements, there is little focus on tools aiding admins/customers to configure the right device mix for their workload or on incorporating algorithms that target dynamic energy savings. EDT addresses these limitations.

Regarding the storage configuration creation problem, systems such as Hippodrome [AHK+02], Minerva [ABG+01], and DAD [ASS+05] address the issue of optimizing storage configuration by iteratively applying several steps such as configuring a low cost storage system, choosing RAID levels and other array parameters, and assigning entire volumes to arrays. EDT's configuration advisor focuses on obtaining the right mix of storage devices to minimize cost is similar to the configuration step in these systems. The key difference is that our approach is inherently aware of, and utilizes the flexibility afforded by EDT's dynamic extent placement.

3.2 Storage Infrastructures

This work is mainly motivated by several proposals on self-managing systems, notably early work on stackable file systems [Ros90] and extensible, self-managing operating systems [BSP+95, GPRA98, SS97], as well as the more recent HP Self-managing Storage project [ABG+01, AHK+02], IBM autonomic computing [KC03] proposal, and CMU Self-$*$ storage systems proposal [Gan03]. In what follows, we examine the related literature.

Block layer virtualization There is abundant work on block layer virtualization from the open source community [dIMO97, Leh99, Pra02, The07] and storage vendors (HP Openview, EMC Enginuity, Symantec/Veritas Volume Manager, and NetApp FlexVol). Typically, these systems allow the administrator to choose from a few pre-existing virtualized configurations (e.g., mapping devices to a logical space and setting up RAID levels). Of the open source initiatives, the GEOM block layer infrastructure is an initial step to what we propose. GEOM classes can perform transformations on I/O requests passing through the block layer. The Violin project [FB05] is similar to GEOM in that it proposes a mechanism for block layer interposition that enables storage virtualization by means of a hierarchy of "virtual devices". While we believe that both GEOM and Violin are steps in the right direction, they still leave a wide chasm to cross for both the developer and the system administrator. GEOM classes and Violin virtual devices must both be implemented from scratch thus fully exposed to the fatalities of kernel-space development. Under these frameworks, extension developers must implement basic block-level I/O primitives themselves (such as cloning, splitting, copying, indirection, replication, etc.) and ensure block consistency in these implementations, thereby leaving significant room for developer error. Further, system administrators must manually consider various possibilities, reason behavior, and configure their storage systems from scratch.

Extensible file systems The framework we propose has similarities at an architectural level to the early 90's work of Rosenthal *et al.* [Ros90] on stackable vnode extensions. Both stackable vnode extensions and block-layer extensions in the framework support seamless extension of the storage stack. WrapFS [Zad99] advocates writing file system extensions as kernel modules and provides a rich support infrastructure.

3.3 Persistent Memory

Persistence techniques can be classified into *system-managed*, *application-managed*, and *application-directed*. System-managed persistence is usually handled by a library with optional OS support. In some solutions, it involves writing a process's entire execution state to persistent storage [GSJ+05, HD06, BMP+04]. Other solutions implement persistently mapped memories for programs with pointer swizzling at page fault time [SKW92]. While transparent to developers, this approach lacks the flexibility of separating persistent and non-persistent data required by many applications and systems software. With application-managed persistence [EP04, OAT+07], application developers identify and track changes to persistent data and build serialization-based persistence and restoration routines. Some hybrid techniques implemented either as persistent memory libraries and persistent object stores have combined reliance on extensive developer input about persistent data with system-managed persistence [CDF+94, Epp89, LAC+96, LLOW91, PBK95, SMK+93]. However, these solutions involve substantial development complexity, are prone to developer error, and in some cases demand extensive tuning of persistence implementations to the storage system making them less portable. For instance, ObjectStore[LLOW91] requires developers to specify which allocations are persistent and their type by overloading the new operator in C++ [obj].

Application-directed persistence provides a middle ground. The application chooses what data needs to be persistent, but a library implements the persistence. The earliest instances were persistent object storage systems [CAC+84] based on Atkinson's seminal *orthogonal persistence* proposal [Atk78]. Applications create objects and explicit inter-object references, and the object storage system (de)serializes entire objects and (un)swizzles reference pointers [Mos92]. Some persistent object

systems (e.g., Versant Persistent Objects [ver], SSDAlloc [BP09], Dali [JLR+94]) eliminate object serialization but they require (varying degrees of) careful development that includes identifying and explicitly tagging persistent objects, identifying and (un)swizzling pointers, converting strings and arrays in the code to custom persistent counterpart types, and tagging functions that modify persistent objects.

Recoverable Virtual Memory (RVM) [SMK+93] was one of the first to demonstrate the potential for memory-like interfaces to storage. However, its approach has some key limitations when compared to the persistent memory library provided under the *Active Block Layer Extensions* (ABLE) project. First, RVM's interface still requires substantial developer involvement. Developers must track all persistent data, allocate these within RVM's persistent region, and ensure that dependence relations among persistent data are satisfied (e.g., if persistent structure a points to b, then b must also be made persistent). Manually tracking such relations is tedious and error-prone. Further, developers must specify the address ranges to be modified ahead of time to optimize performance. These requirements were reported to be the source of most programmer bugs when using RVM [MSSL97]. Second, RVM's static mapping of persistent memory segments makes it too rigid for contemporary systems that demand flexibility in managing address spaces [LNBZ08, The09]. In particular, this approach is not encouraged in today's commodity operating systems that employ address-space layout randomization for security [BDS03, The09]. Finally, RVM is also restrictive in dynamically growing and shrinking persistent segments and limits the portability of a persistent segment due to its address range restrictions.

The recent Mnemosyne [VTS11] and NV-Heaps [CCA+11] projects also provide persistent memory abstractions similar to ABLE. However, there are at least two key differences. First, both of the solutions are explicitly designed for non-volatile

memories or NVM (e.g., phase-change memory) that are not yet commercially available. Most significantly, these devices are intended to be CPU accessible and byte addressable which eliminates copying data in/out of DRAM [CNF+09]. Thus, the focus of these systems is on providing consistent updates to NVM-resident persistent memory via transactions. On the other hand, ABLE targets currently available commodity technology. Second, neither of these systems provide the *orthogonal persistence* that ABLE enables; rather, they require the developer to explicitly identify individual allocations as persistent or not and track and manage changes to these within transactions. For instance, the NV-Heaps work argues that explicit tracking and notification of persistent data ensures that the developer does not inadvertently include more data than she intends [CCA+11]. We take the converse position that besides making persistence vastly simpler to use, automatic discovery ensures that the developer will not inadvertently exclude data that does need to be persistent for correctness of recovery, while simultaneously retaining the ability to explicitly exclude portions of data when unnecessary. Further, ABLE's design, which relies on interposing on application memory allocations, ensures that pointers to library structures (e.g., files or sockets) are reset to NULL upon container restoration by default, thus relieving the developer of explicitly excluding such OS dependent data; such OS specific data is typically re-initialized upon application restart. Finally, feedback about automatically discovered persistent containers from ABLE can help the developer in reasoning about and eliminating inadvertently included data.

Single level persistent stores as used in the Grasshopper OS [DdBF+94] employ pointer swizzling to convert persistent store references to in-memory addresses at the page granularity [VD92, WD92] by consulting an object table within the object store or OS. Updates to persistent pointers are batch-updated (swizzled) when writing

pages out. ABLE fixes pointer addresses when persistent containers get loaded into memory but is free of swizzling during container writing time.

Finally, Java objects can be serialized and saved to persistent storage, from where it can be later loaded and recreated. Further, the Java runtime uses its access to the object's specification, unavailable in other lower-level imperative languages that ABLE targets.

CHAPTER 4

BLOCK LAYER EXTENSIONS

In this chapter we present our experiences developing storage extensions. We showcase three extensions. BORG which optimizes the data layout to reduce I/O latencies based on observed access patterns. EXCES which uses an external caching device to place frequently accessed data in an effort to reduce I/Os to the hard disks to save power. And finally, EDT, a dynamic multi-tier system that leverages the benefits of each storage tier by placing data in the tier where its best served according to observed I/O characteristics.

After developing these extensions we found that there is a substantial amount of auxiliary functionality shared by all three. For instance, in all cases we required a mechanism to correctly migrate data, either when creating the new layout in BORG or when moving data between devices in the case of EXCES and EDT. Doing this correctly requires a delicate ordering of I/O operations. Many complications could arise when the migration is underway, which we will discuss later.

The rest of this chapter is structured as follows, we start by motivating the development of self-optimizing storage extensions at the block layer. Next, we present three self-optimizing storage extensions we have built, namely, BORG (§4.2) which optimizes data layout based on observed I/O patterns, EXCES (§4.3) which lowers disk power consumption by caching popular blocks on a low power device, and EDT (§4.4) a two component system which helps design and deploy multi-tier storage systems that improve performance and save energy. We end with a summary of the lessons we take away from the developing these extensions.

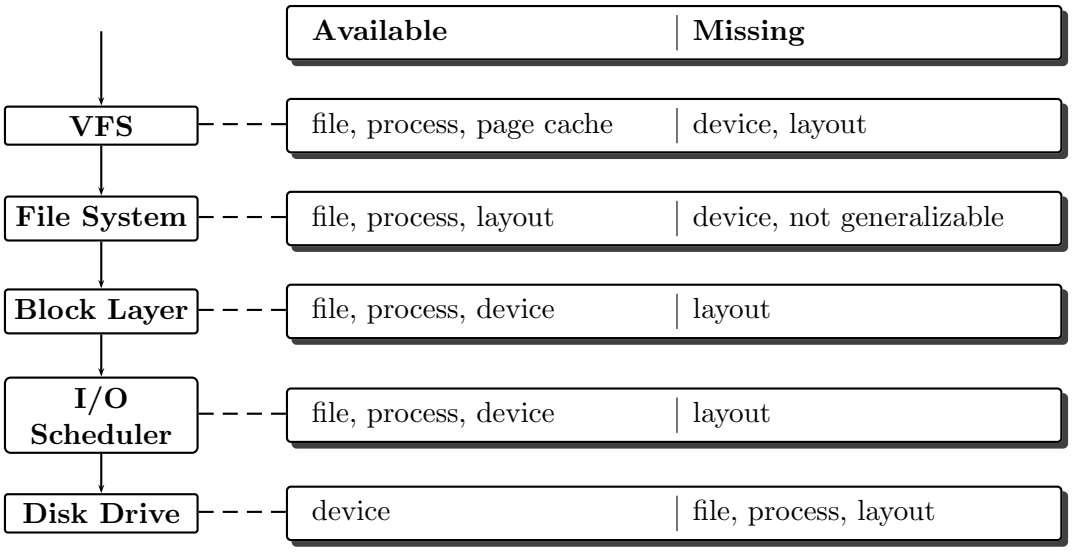


Figure 4.1: Storage Layers and information available.

4.1 A Block Layer Solution.

Our choice of the block layer as the building block for the storage system extensions is prompted by several requirements both from a development and administration stand-point. First, numerous solutions in the research literature attest that the block layer provides a conducive environment for storage self-management, starting from well-established software RAID implementations [dIMO97, Cou96] to several recent optimizations [ASV06, FMK+07, GPK+07, LCSZ04, MAC+08, Nar08, NDT+08, QD02, SWS05, ZLP08]. Second, the block abstraction is a prevalent abstraction in storage architectures. Figure 4.1 summarizes the storage stack layers of most systems as well as the information available in each layer. Notice that for network-attached SAN [Phi98] devices, the block layer includes the bottom layer at the initiator and the top layer of the target. For NAS [GV00] and object-based storage [MGR03], this layer resides within the target's OS, as in stand-alone storage. Third, several requirements for a self-management infrastructure building-block, from both developer and administrator stand-points, are satisfactorily met as listed below:

1. *Access to system state* Self-management solutions rely on sensing the system state continuously. The block layer provides *temporal*, *process-level*, and *block-level* attributes for each block I/O request as well as device and I/O state, including failed I/O operations or devices.

2. *A simple interface* The block layer virtualizes storage as a logical block address space and adheres to a simple block consistency contract which requires that a block read contains the exact same data that was last written to it. A functionally simple layer allows arbitrarily complex self-management extensions to be built on top of it as long as the extensions adhere to its interface and contract. Most importantly, this simplicity in semantics also allows us to build a theory of block layer extensions.

3. *File system independence* The block layer enables self-management solutions that can work with multiple heterogeneous file systems without changes to their implementations. This gains importance given the variety of designs and implementations of file systems.

4. *File system accessibility* A block layer development infrastructure can be easily made available to the file system instance(s) above it. Thus, file system layer innovations can tap into ABLE infrastructure for support. We elaborate on this further in § 5.2.3.

5. *I/O scheduler leverage* Self-management extensions may generate additional I/O traffic which automatically leverage scheduling optimizations such as request merging and ordering. This feature frees the developer from generic concerns of I/O handling, allowing her to focus on developing the functional core of the extension.

6. *Control and arbitration of storage resources* Due to its position in the storage stack, the block layer can easily serve as an arbitrator block storage resources. For instance, self-management extensions can request and use dedicated storage, distinct from file system volumes or allocations of other ABLE extensions.

Caveat Building self-management extensions at the block layer has a strong list of advantages, but by no means we suggest the block layer as the ideal building block for all self-management storage extensions. Extensions that closely depend on the semantic properties of other layers are probably best suited elsewhere in the stack. However, as evidenced by recent research, we do claim that a large set of storage solutions can be implemented most conveniently at the block layer.

Now that we have argued for the block layer as our preferred place in the storage stack for extension development, we discuss the design of three novel extensions.

4.2 Optimizing Data Layout for Performance

Present day file systems, which control space allocation on the disk drive, employ static data layouts [HD96, KBBA, MJLF84, Nam, Twe98, Cus94]. However if file systems were able to adapt and optimize to the dynamic characteristics of I/O workload performance could be greatly improved.

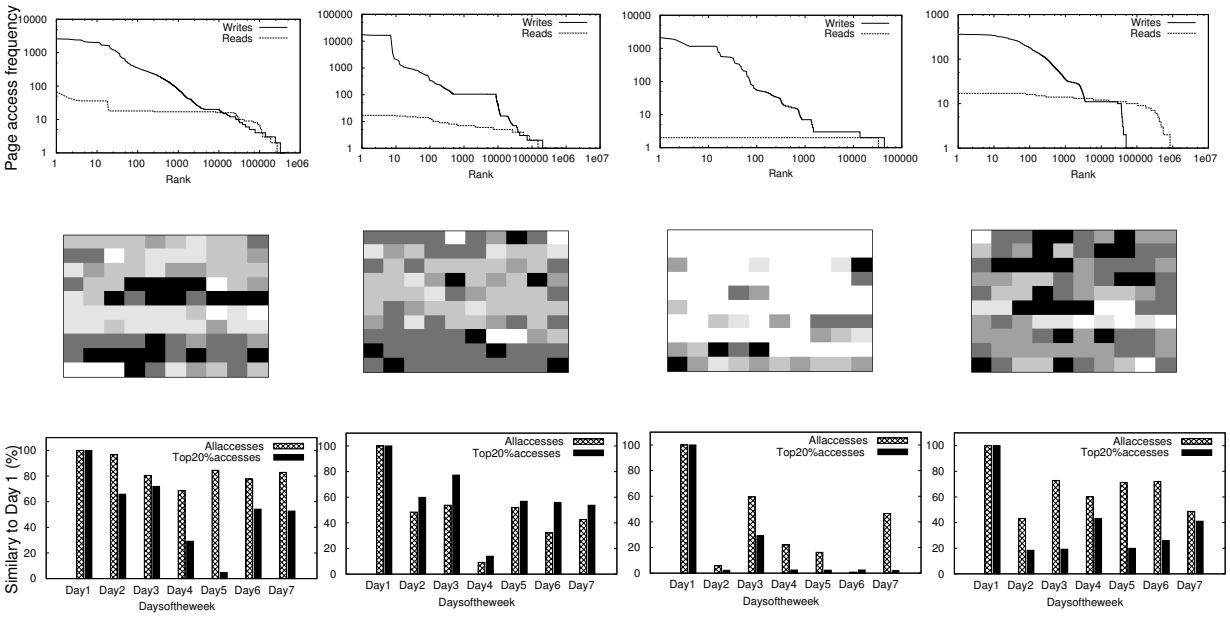
To validate this hypothesis we conducted experiments to reconcile past observations about the nature of I/O workloads [RW93a, GS02, HSY05] in the context of current-day systems including end-user and server-class systems, Table 4.1 presents a summary of the workloads studied, more details on the workloads can be found in [BGU+09]. Our key observations are: *(i)* on-disk data exhibit a *non-uniform access frequency distribution*; the "frequently accessed" data is usually a small fraction of the total data stored when considering a coarse-granularity time-frame. For

the traces that we collected where less than 4.5-22.3% of the file system data were accessed over the duration of an entire week (shown in Table 4.1). Further, the top 20% most frequently accessed blocks contributed to a substantially large (∼45-66%) percentage of the total accesses across the workloads depicting a skewed data access frequency. *(ii)* considering a fine-granularity time-frame, the "on-disk working-set" of typical I/O workloads is dynamic; nevertheless, workloads exhibit *temporal locality* in the data that they access. Figure 4.2 (bottom row) depicts the changes in the per-day working-sets of the I/O workload. The two end-user I/O workloads and the web server workload exhibit large overlaps in the data accessed across successive days of the week-long trace with the first day of the trace. *(iii)* I/O workloads exhibit *partial determinism* in their disk access patterns; besides sequential accesses to portions of files, fragments of the block access sequence that lead to non-sequential disk accesses also repeat. Table 4.1, we present the *partial determinism* for each workload calculated as the percentage of non-sequential accesses that repeat at least once during the week. The partial determinism percentages are high for the two end-user and the SVN server workloads.

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]		Writes [GB]		File System accessed	Top 20% data access	Partial determinism
			Total	Unique	Total	Unique			
office	8.29	1.5	6.49	1.63	0.32	0.22	22.22 %	51.40 %	65.42 %
developer	45.59	2.0	3.82	2.57	10.46	3.96	14.32 %	60.27 %	61.56 %
SVN server	2.39	0.5	0.29	0.17	0.62	0.18	14.60 %	45.79 %	50.73 %
web server	169.54	0.5	21.07	7.32	2.24	0.33	4.51 %	59.50 %	15.55 %

Table 4.1: Summary statistics of week-long traces obtained from four different systems.

Based on the above observations, we believe that dynamically optimizing the disk layout has potential to improve I/O performance, given that: *(i)* it is reasonable to expect that co-locating frequently accessed data in a small area of the disk would help reduce seek times [AS95]. *(ii)* optimizing layout based on past I/O activity can



(a) office (b) developer (c) SVN server (d) web server

Figure 4.2: Rank-frequency, heatmap, and working-set plots for week-long traces from four different systems. The heatmaps (middle row) depict frequency of accesses in various regions of the disk, each cell representing a region. Six normalized, exponentially-increasing heat levels are used in each heatmap where darker cells represent higher frequency of accesses to the region. Disk areas are mapped to cells in row-major order.

improve future I/O performance for some workloads. *(iii)* there is ample scope for optimizing the repeated non-sequential access patterns.

BORG is an online *Block-reORGanizing* storage system to comprehensively address the above issues. BORG correlates disk blocks based on block access patterns to capture the I/O workload characteristics. It dynamically copies working-set data blocks (possibly spread over the entire disk) in their relative access sequence contiguously to a *BORG OPtimized Target (BOPT)* partition, thus simultaneously reducing seek and rotational delays. In addition, it assimilates all *write requests* into the BOPT partition's write buffer. Since BORG operates in the background it presents little interference to foreground applications. Next, we present its design.

4.2.1 BORG System Architecture

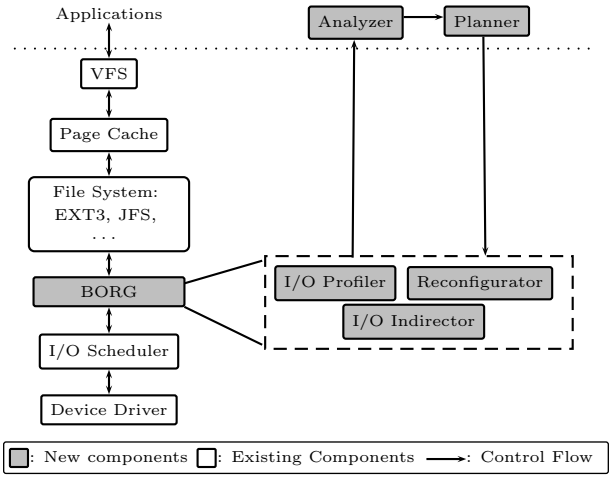


Figure 4.3: BORG System Architecture.

Abstractly, BORG follows a four-stage process:

1. *profiling* application block I/O accesses,
2. *analyzing* I/O accesses to derive access patterns,
3. *planning* a modification to the data layout, and

4. *executing* the plan to reconfigure the data layout.

In addition, an I/O indirection mechanism runs continuously re-directing requests to the partition that it optimizes as required. Figure 4.3 presents the architecture of BORG in relation to the storage stack within the operating system. The modification to the existing storage stack is in the form of a new layer, which we term *BORG layer*, that implements three major components: the *I/O profiler*, the *BOPT reconfigurator* and the *I/O Indirector*. A secondary throttle-friendly user-space component implements the *analyzer* and the *planner* stages of BORG and performs computation and memory-intensive tasks. While profiling and indirection are both continuous processes, the other stages run periodically and in succession culminating in a reconfiguration operation.

For the I/O profiler, we use a low-overhead kernel tool called `blktrace` [Axb07]. The analyzer reads the I/O trace collected by the profiler periodically (based on a configurable reconfiguration interval) and derives data access patterns. Subsequently, the planner uses these data access patterns and generates a new reconfiguration plan for the BOPT partition, which it communicates to the BOPT reconfigurator component. The user-space analyzer and planner components run as a low-priority process, utilizing only otherwise free system resources. Under heavy system load, the only impact to BORG is that generating the new reconfiguration plan would be delayed.

The BOPT reconfigurator is responsible for the periodic reconfiguration of the BOPT partition, per the *layout plan* specified by the planner. The reconfigurator issues low-priority disk I/Os to accomplish its task, minimizing the interference to foreground disk accesses. Finally, the I/O indirector continuously directs I/O requests either to the FS partition or the BOPT partition, based on the specifics of the request and the contents of the BOPT.

4.2.2 BORG Evaluation Results

BORG was evaluated and shown to offer performance gains in the average case for varied workloads including office and developer class end-user systems, a web server, an SVN server, and a virtual machine monitor; we present a summary of those results below; please refer to the full BORG paper for more detailed results and findings [BGU+09]. Average disk busy times reduction with BORG across these workloads range from 6% (for the VM workload) to 50% (for the developer server workload). Occasionally BORG performs worse than a vanilla system, specifically when a read-mostly workload (e.g., a web server) drastically shifts its working set. BORG is able to easily address changing working-sets with a (possibly non-sequential) write workload (e.g. an SVN server), since it has the ability to absorb and sequentialize writes inside the BOPT. A sensitivity analysis revealed the importance of choosing the right configuration parameters for reconfiguration interval, BOPT size, and the write-buffer fraction. Fortunately, simple iterative algorithms can be quite effective in identifying the right parameter combination; a formal investigation of such an approach is an avenue for future work. The memory and CPU overheads incurred by BORG are modest and with ample scope for further optimization.

4.2.3 BORG Summary

BORG is a self-optimizing layer in the storage stack that automatically reorganizes disk data layout to adapt to the workload's disk access patterns. It was designed to optimize both read and write traffic dynamically by making reads and writes more sequential and restricting majority of head movement within a small optimized disk partition. BORG offers a novel and practical approach to building self-optimizing

storage systems that can offer large I/O performance improvements in commodity environments.

Researchers have suggested layout optimizations that reduce I/O latencies can contribute to power savings since they reduce the time the disk is busy [HHS05]. However, we feel it is best to use low power storage devices as a cache for disk and thereby amplify for the potential for energy savings. We discuss an extension which implements this below.

4.3 Lowering Power Consumption with External Caching

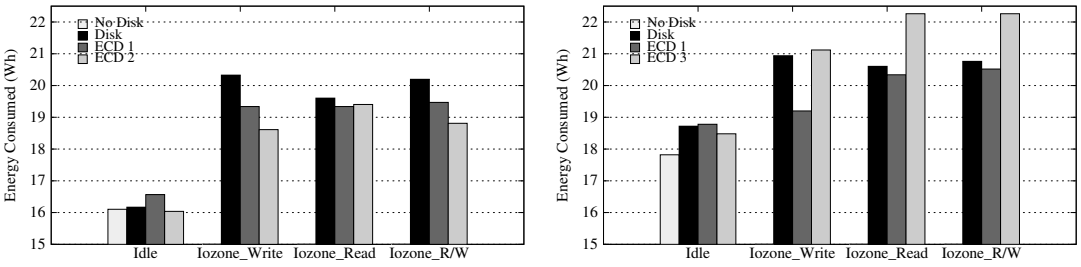
The next extension we discuss tackles the issue of build energy-efficient storage systems for personal computing. The key argument is that the disk drive, the sole mechanical device in modern computers, is also one of its most power consuming [Int02]. Previous simulation based work [BBL06, CJZ06, MDK94] suggests using a power lower non-volatile storage device [IBM, Tec], which refer to *external caching device* (ECD), as a cache to the hard disk and spindown the disk during idle periods to save power. 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 power consumption of the system as a whole, but only focus on the reduction in disk power consumption. Second, existing studies do not evaluate an important artifact of external caching, which is the impact on application performance. 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.

Config.	Disk State	Iozone Data	ECD Specification	ECD Interface
No Disk	Standby	N/A	N/A	N/A
Disk	Active	On disk	N/A	N/A
ECD 1	Standby	On ECD	SanDisk Cruzer Micro USB	USB interface
ECD 2	Standby	On ECD	SanDisk Ultra CF Type II	eFilm Express Card 34 CF Adapter
ECD 3	Standby	On ECD	SanDisk Ultra CF Type II	SanDisk Ultra PC Card Adapter

Table 4.2: Various laptop configurations used in profiling experiments.

To understand the power 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 (see Table 4.2). We measured the overall system power 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 4.4 depicts the individual power consumption profiles for each storage device on two different laptops: `shiriu` and `beer`. A detailed experimental setup is given in [UGB+08]. We see that both types of flash memory consume less power than the disk in all the Iozone benchmarks, except for ECD 3. More importantly, for both systems, even in configurations when the disk is powered down completely, we observe that the power savings are bound within 10% for an I/O intensive benchmark. Further, when the system is idle, the ECD subsystems consumes as much power 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 power. Our goal in this study is to address this question comprehensively.

We present EXternal Caching system for Energy Savings (EXCES), it 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 saving power. EXCES adapts to workload changes by identifying popular data continuously, reconfiguring



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

Figure 4.4: Power consumption profiles of various ECD types and interfaces.

the contents of the ECD (as and when appropriate) to maximize ECD hits on both read and write operations. To prefetch popular data to 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 perform 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.

4.3.1 EXCES: System Architecture

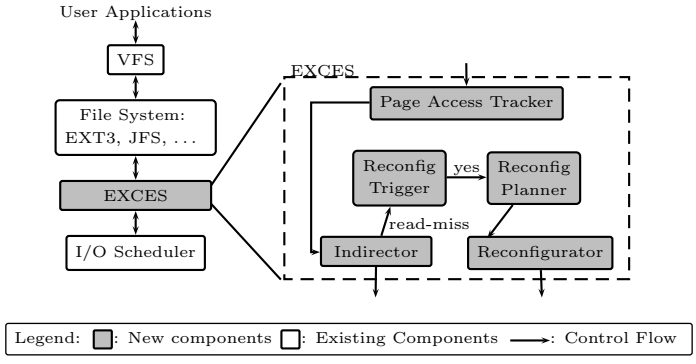


Figure 4.5: EXCES system architecture.

EXCES consists of five major components as shown in Figure 4.5. 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.

4.3.2 EXCES Summary

EXCES is an external caching system that reduces system power consumption by prefetching, caching, and buffering disk data on a less power 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 the EXCES system 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.

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. Optimizations that address random write performance on the ECD will gain significant importance in such systems. Recently released flash based solid state disks (SSD), such as the Intel® X-25m, partially address the random write problem using an intelligent translation layer. In the next section we look into a system that leverages this newer flash based devices.

4.4 Improving Performance with Dynamic Multi-tier Systems

Previously in § 4.3 we designed an external caching system for low end flash devices. We saw the EXCES system was able to achieve modest power savings, but could also suffer from performance degradation due mostly to random writes. In this work, we continue on to multi-tier systems, those that incorporate multiple types of storage devices. These systems strive to provide performance and reliability at minimum capital and operating cost. Typically, these systems use high performance disk drives (e.g. SCSI/SAS/FC) to provide that performance. However, solid-state drives (SSDs) offering superior random access capability per GByte have become increasingly affordable. On the other hand, SATA drives offering superior cost per GByte are also attractive for mass storage. Systems with only SSDs are still too expensive, and those built using only SATA would not provide enough performance/GByte for most enterprise workloads. Multi-tier systems containing a mix of devices can provide high performing and lower cost storage by utilizing SSDs only for the subset of the data that needs SSD performance.

Current commercial SSD-based multi-tier systems from Compellent [Pet09a], IBM [Tan10], 3PAR [Pet10], and EMC [Lal09] provide performance gains and cost

savings. However, customer adoption has been slow. One of the reasons for this is the difficulty in determining what mix of devices will perform well at minimum cost in the customer's data center. This optimization task is highly complex because of the number of device types available along with the variability of workloads in the data center.

To address this challenge, two things are needed: configuration tools to assist in building such systems and to demonstrate potential benefits based on customer workload, and capabilities in the storage systems that can optimize placement of data in the tiers of storage. The placement should ensure that actively accessed data is co-located to minimize latency while lightly accessed data is placed most economically. There is also an opportunity to improve operating cost by placing data on the minimum set of devices that can serve the workload while powering down the rest. Current products address some but not all of these challenges. Determining which mix of devices to buy remains a difficult problem, and improvement of operating cost by consolidation and power management has not yet been tackled.

To address these gaps, we develop an Extent-based Dynamic Tiering (EDT) system that includes: 1) a Configuration Adviser tool *EDT-CA* to calculate cost-optimized mixes of devices that will service a customer's workload, and 2) a Dynamic Tier Management *EDT-DTM* component that runs in the configured storage system to place data by *dynamically* moving *extents* (fixed-size portions of a volume) to the most suitable tiers given current workload. EDT-CA works by simulating the dynamic placement of extents within tiers that offer the lowest cost to meet an extent's I/O requirements as they change over time, and thus suitably size each tier. EDT-DTM monitors active workload and manages extent placement and migration in such a way that performance goals are met while optimizing operating cost where

feasible by consolidating data into fewer devices within each tier and powering off the rest.

This work makes the following contributions:

- EDT is the first publicly available work that formalizes and explores the design space for storage configuration and dynamic tier management in SSD-based multi-tier systems. (§ 4.4.1)
- EDT consists of a novel configuration algorithm for dynamic tiered systems that outputs lower cost configurations. (§ 4.4.2, § 4.4.3)
- EDT proposes a novel dynamic placement algorithm to satisfy performance requirements while minimizing dynamic power. (§ 4.4.4)
- EDT outperforms SAS-only and other simpler extent-based tiering approaches across a variety of workloads in both cost and power. (§ A.1)

4.4.1 Multi-Tiering Design Choices

This section describes important design choices for a multi-tier system that enable efficient use of the tiers.

Extent-based Tiering

The first we consider the granularity of data placement. As we saw in § 4.2 and previous studies [GPG+09], I/O activity is highly variable across LBAs in a volume. Therefore, if data were placed at a volume level based on average volume workload characteristics, a large percentage of the tier will hold data that does not require the tier's capabilities. Thus, we perform data placement at the granularity of an **extent**, a fixed-size portion of a volume. The smaller the extent size, the more efficient will be the data placement. However, the amount of metadata overhead

required to keep track of extent locations and other statistics increases as extent size is decreased. We choose an extent size with an acceptable system overhead (details in § A.1.2).

Dynamic Tiering

Regarding the time scale at which extents move across tiers, one choice involves placing extents once during system instantiation or moving them at coarse grain intervals of the order of days or months. However, studies show that I/O rates of a workload are typically below peak most of the time [KR10, LPGM08], this static or semi-static placement is not optimal —a placement that configures for the peaks pays extra in both cost and energy for a system that is over-provisioned at off-peak times; and a placement that mitigates cost from over-provisioning by configuring for the average I/O rate suffers from decreased performance during peaks.

The alternate choice is to plan extent movement at intervals on the order of minutes or hours. We refer to this time interval as an **epoch**. Such a system exploits variation in extent I/O rate to improve its efficiency; an extent is on a SATA tier when inactive, and moves to the SAS or SSD tier as its I/O rate goes up. This achieves cost-effective use of resources and/or dynamic energy savings. Similarly, when the performance demanded of a single tier is below its peak capacity, extents placed on the tier can be consolidated into fewer devices for power savings. Often, the set of heavily loaded extents changes over time [GPG+09]. Dynamic migration of the heavily loaded extents into SAS or SSD when required enables cost-effective use of the resources. Thus, we choose to perform dynamic data placement with an epoch length of the order of minutes/hours.

The drawback of such a dynamic system, however, is the cost of data migration, i.e., the potential adverse effect on foreground I/O latency and the migration

latency itself before the desired outcome. Longer epoch durations allow more time to execute migrations and amortize overhead better. Thus, we pick an epoch duration whose estimated migration overhead is below the allowable system migration overhead (details in § A.1.2). Additionally, it is important to ascertain that the overhead of migrating data does not overwhelm its benefit. This depends on the stability of the workload—extents that relocate often benefit less from migration compared to extents that stay longer in a particular tier. The workloads we have studied indicate that dynamic migration is typically beneficial, but we believe that a dynamic system must also be able to back off when lack of workload stability causes dynamic migration to interfere with performance.

Beyond I/O Rate Based Tiering

This design choice determines the extent-level statistics required to match an extent with the right tier. The available public documentation about commercial extent-based multi-tier products indicates use of IOPS to measure load; in these systems high IOPS regions are placed onto SSD while leaving the remainder of the data on SAS or SATA. Although this method is intuitively correct, our preliminary analysis reveals significant drawbacks: IOPS-based placement does not factor in the bandwidth requirement of an extent. For example, consider an extent with a long sequential access pattern consisting of small I/Os to contiguous locations. Such an extent will have high IOPS and bandwidth requirements. Our analysis of production and SPC-1 [spc] like workload traces (§ A.1), collected after the I/O scheduler show such patterns. Using I/O rate statistics for this stream causes sequential streams, which are more cost-effectively served on SAS or even SATA, to be inappropriately placed on SSD. IOPS placement also ignores capacity of the extent. An extent

with high IOPS relative to other extents may not have high enough I/O density (IOPS/GByte) to justify the high $/GByte cost of the SSD.

Our approach is to collect more than just I/O counts. We employ a heuristic as in [NTD+09] to break down an extent's workload: I/Os that access LBAs within 512 KBytes of the previous ones are taken as part of a sequential stream and contribute to an extent's bandwidth requirement. I/Os further apart are characterized as *random* I/Os and are used to compute a random I/O rate. Thus, for each extent, we collect a random I/O rate and bandwidth. Other methods for separating the I/Os into random and sequential may also be applicable.

4.4.2 Design Overview

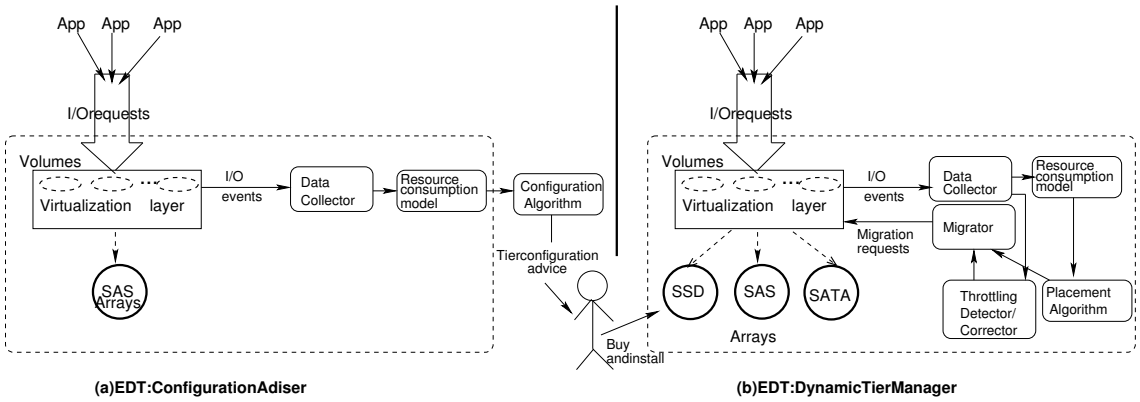


Figure 4.6: EDT system architecture.

EDT consists of two elements as depicted in Figure 4.6: a Configuration Adviser (EDT-CA) that determines the right number of devices per tier to install into a storage system, and a Dynamic Tier Manager (EDT-DTM) that operates inside a running system and continuously manages extent placement across tiers. EDT is expected to be deployed in a commercial storage system as shown in Figure 4.6 which exports many volumes, includes a virtualization layer that allows volumes

to be made up of extents stored in arrays of different device types, is capable of collecting and exporting statistics about extent workloads, and can execute requests to non-disruptively move extents between storage devices.

An example usage scenario is as follows: A user wishes to replace a SAS based storage array with a new, tiered storage system with twice the capability. He collects a trace of his workload over a 24 hour period that he thinks is representative. The trace is then run through EDT-CA which produces the minimum cost configuration of SSD, SAS, and SATA that can provide 2x the performance of the existing system. EDT-CA is aware of the runtime migration capabilities of EDT-DTM and takes them into account when determining the configuration. The user installs the new system. During operation of the new system, EDT-DTM manages migration between tiers by continuously collecting extent level statistics, consolidates data onto lower-power tiers when possible, and monitors the system to ensure that the workload performance is not throttled.

In general, EDT-CA starts by determining the workload requirements for the system it is going to configure. This can either be done with a user generated general description of requirements including IOPS, seq/random mix, length of I/O requests, and their distribution across extents, or by using time series data collected from a workload running on an existing system. For the scope of this work, we assume availability of time series statistics. In this approach, EDT-CA takes a epoch-granularity trace of extent workload statistics sampled at times when storage system usage is high. It then estimates the resources required in different tiers to satisfy that workload by simulating placement of each extent in a tier that minimizes its incurred cost while meeting its performance requirements. It repeats this process every epoch and assigns extents to their lowest cost tier based on their performance requirements in that epoch. At the end of this simulation, EDT-CA determines the

set of devices that are needed based on the maximum number of devices needed in each tier over all the epochs. This configuration determines the set of devices purchased by the user.

Once the new tiered system is up and running, EDT-DTM manages extent placement. It collects extent level statistics, estimates extents' resource consumption in different tiers, and then plans and executes migrations. EDT-DTM implements a throttling correction mechanism to ensure that performance requirements are satisfied as they vary over time; it constantly monitors array performance and if performance throttling is detected relocates extents to restore performance. EDT-DTM's placement algorithm seeks to place each extent into the lowest-energy tier that satisfies its performance requirement and then to further minimize energy by consolidating extents in the same tier into fewer devices allowing unused devices to be powered down. Both these algorithms use a Migrator module to move extents.

EDT-CA and EDT-DTM work together to minimize cost. EDT-CA minimizes acquisition cost, and EDT-DTM minimizes operating cost. As our results will show, configurations based on static extent placement are more expensive both to acquire and operate.

Common Components

EDT-CA and EDT-DTM share components that collect statistics and calculate resource consumption.

Data Collector The *Data Collector* receives information about I/O completion events including the transfer size, response time, logical block address (LBA) , the volume ID to which the I/O was issued, and the array which executed the I/O. The collector then maps the (LBA, volume id) pair of each I/O to a unique extent

in the system, and compiles for each extent, the number of random I/Os and the number of transferred bytes. It then periodically (every minute in our implementation) computes instantaneous bandwidth and random IOPS per extent as well as an exponentially-weighted moving average. In addition to the extent statistics, the collector aggregates statistics per array. It maps each I/O to its array and compiles its IOPS and average response time. These measurements are used by EDT-DTM to determine if I/Os on an array are being throttled. For a very large system the amount of data collected by the data collector may be significant. If this is an issue, the the extent size can be made larger to reduce the volume of statistical data.

Resource Consumption Model The *Resource Consumption Model* uses the extent statistics to estimate the resources it consumes when placed on a device of a given type. Resources are allocated based on the observed capacity and performance requirements at the device level. Therefore, any workload optimizations like deduplication, compression, and caching do not need to be considered in these models as their effects will be captured by the usage statistics.

An extent consumes the resources of a device along capacity and performance dimensions. Consider an extent of size E_c and a performance requirement E_p determined by its random IOPS rate ($RIOR$) and bandwidth measured in previous epochs. The fraction of capacity required to host an extent E in device D ($RC(E_c, D)$) is straightforward:

$$RC(E_c, D) \quad = \quad \frac{\text{Capacity required by extent}}{\text{Total space in device}}$$

For performance utilization, we use a simplified model based on Uysal *et al.*'s work [UAM01]. The performance resource consumption of extent E, when placed on device D ($RC(E_p, D)$) is:

$$RC(E_p, D) \quad = \quad \text{RIOR} \cdot \text{Rtime} + \text{Bandwidth} \cdot \text{Xtime}$$

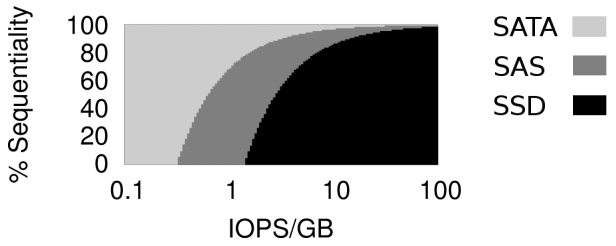


Figure 4.7: Lowest cost tier for extents with different characteristics.

Here *RIOR* is the number of random I/Os sent to an extent in a second (IO/s) and Rtime is the expected response time of the device (s/IO). *Bandwidth* is the bandwidth requested from the device (MB/s), and *Xtime* is the average transfer time (s/MB). The result of this equation is the fraction of the device performance utilized by an extent. Note that the Rtime and Xtime values are averages and may need to be adjusted depending on the expected workload. For example an SSD with a mostly random write workload would have significantly higher Rtime than the same SSD with a mostly random read workload. The overall resource required by an extent is then the maximum of the capacity utilization fraction and the performance utilization fraction:

$$RC(E, D) \quad = \quad max(RC(E_p, D), RC(E_c, D))$$

The resource consumption model determines the most efficient tier for an extent. For instance, when minimizing cost, the most suitable tier is the one where the extent incurs the lowest cost (the product of the device cost and the extent's resource consumption on that device). Figure 4.7 confirms the advantage of multi-tier systems since the most cost-effective tier changes with extent characteristics, namely the total IOPS and the percentage of sequential accesses among three classes of storage devices specified in Section A.1. As expected, we observe that mostly idle extents favor SATA, medium IOPS favor SAS, and high IOPS favor SSD. Further, as expected, more sequential extents favor HDDs.

4.4.3 Configuration Adviser

EDT-CA builds on the Data Collector and the Resource Consumption Model described above. Since configuration is an NP-Hard packing problem, we propose a light-weight heuristic to achieve low cost extent placement:

1. **Binning.** For each extent E, and device type D, we compute the cost of allocating the extent to that device as extent cost(E, D) = cost(D) $\cdot$ RC(E,D). The extent is then placed in the tier that meets its performance with the lowest cost. Iterating over all the extents, the above computation separates the extents into bins, one per each tier.

2. **Sizing a bin.** For each bin, we obtain its performance and capacity resource consumption as $RC_p = \sum RC(E_p, D)\ \forall E$, and $RC_c = \sum RC(E_c, D)\ \forall E$.

   The maximum of these two values gives the total bin resources required, and the number of required devices of this bin type are computed by rounding up this sum to the nearest integer value.

3. This process is independently repeated for each epoch to identify the number of devices per tier that yields minimum cost for that epoch.

4. The last step consists of combining these different configurations to obtain a final system configuration valid across time. For the scope of this work, we achieve the final configuration by allocating the maximum number of devices of each type used across all epochs. That is, if at epoch t_0 2 devices of type D and 1 of type D' are the most cost effective, but at epoch t_1 1 of type D and 2 of D' is better, then our method will indicate that we need 2 of type D and 2 of D'.

Our current method of combining configurations across epochs is fairly conservative and could potentially result in an over-provisioned system. However, as our current algorithm already results in lower cost configurations (Section A.1), we relegate exploring more efficient ways of combining configurations over time to future work. Also note that when we compute tiered configuration for each epoch independently, we assume that the extents can be suitably migrated between epochs if required. As part of our future work, we intend to model the required number of migrations, and suitably adjust the provisioning if the required migrations exceed the maximum number of migrations a system can support in a chosen interval of time. Finally, our Configuration Algorithm can also be used to upgrade a multi-tier system to meet upcoming performance demands.

4.4.4 Dynamic Tier Manager

EDT-DTM combines three new modules with the Data Collector and the Resource Consumption Model to continuously optimize extent placement: (1) a Tiering and Consolidation module, (2) a Throttling Detector/Corrector module, and (3) a Migrator module.

Tiering and Consolidation Algorithms

At the end of every epoch, the Tiering and Consolidation (TAC) algorithms generate an extent placement to satisfy extent performance requirements and minimize dynamic system power. Such an energy efficient placement can be achieved both by leveraging the strengths (i.e. performance or capacity per watt) of the heterogeneous underlying hardware (SSD, SAS, and SATA drives), and by consolidating data into fewer devices when possible and turning off the unused devices.

Similar to the configuration problem, placement for power minimization is also NP-Hard, and we propose a heuristic solution. TAC requires two inputs: (1) current random I/O rate and bandwidth for each extent from the actively running system, and (2) size (in bytes) and the random I/O rate and bandwidth capability for each array in the storage system. It then uses a two-step process to output a new extent placement that aims to adapt to the changes in the workload as follows:

(1) **Tiering.** For each extent E, and device type D, we compute the "fractional power burden" of allocating the extent to that device as extent power(E, D) = power(D)·RC(E,D). The extent is then placed on the tier that meets its performance with the lowest power consumption. Doing so allows EDT to reduce active power via consolidation (described next). Iterating over all the extents results in one bin per tier. The assignment of extents to a tier is performed locally on an extent by extent basis, irrespective of the total performance needs or available space in that tier.

(2) **Consolidation.** Extents assigned to each tier are then sorted using their RC values and placed in arrays using the First Fit Decreasing heuristic, a good approximation algorithm to the optimal solution for extent packing [Yue91]. When extents already assigned to the tier under consideration exceed its available performance (i.e., resource consumption metric for the assigned extents exceeds 1) or the tier runs out of space in the available arrays, the remaining extents in the extent list are demoted to the tier with the next lower power burden for that extent. This packing process is now repeated for all the tiers, consolidating extents into a minimum number of arrays in a tier. Extents already in the right tier and on an array that will remain powered on in this epoch retain their position from the previous epoch, thereby saving migrations. Any unused arrays from the extent placement are set to a lower power state to conserve energy.

Throttling Detector and Corrector

While the TAC mechanisms enable dynamic performance and power optimization, unexpected load and working set changes can suddenly alter the performance requirements of extents. However, tracking this performance change, especially when an extent's I/O rate increases, is challenging. Extents placed in a low performance tier cannot exhibit high I/O rates even when the application above may desire it. This causes *throttling* of the true IOPS requirement of the extent, artificially limiting it to a low value. The Throttling Detector overcomes this limitation by monitoring the average response time of each active array every minute.

If the average response time of I/Os from an array indicates that undesirably high request queuing is occurring in the array, EDT decides that the array is throttling the true IOPS requirement of applications and causing delays. When throttling is detected, pending migrations driven by TAC are immediately halted and EDT-DTM switches to a *throttling correction* mode to perform recovery. To respond rapidly and minimize the possibility of future throttling in the same array, the load on the throttled array is shed by migrating a minimum set of extents responsible for at least half of its current total performance resource consumption.

To select the target array(s), we first start by considering the best possible tier for each extent being migrated, and within that tier we first examine arrays which are already active to see if they can absorb the new extent. If none can host the new extent, we consider arrays that are not in use in that tier if any are available. If the best tier can not accommodate the extent we try the same approach on tiers with the next higher power burden for that extent. If the array continues to remain throttled after half the load on the array has been migrated, the extent migration process is repeated, until the system is no longer throttled. The entire system stays in recovery mode while an array remains throttled, suspending energy optimizing

migrations. When no arrays are throttled, the system switches back to the TAC placement after an epoch elapses.

Migrator

The Migrator handles the data movement requests from TAC and the throttling algorithms. It compares the new placement of the extents from the above algorithms to their old placement, and identifies extents that need to be migrated. It then schedules and optimizes these migrations. On one hand, migrations that relieve throttling must be completed quickly. On the other hand, migrations cause additional I/O traffic, and care must be taken so that they do not affect the foreground I/O performance.

Our migration scheme achieves this tradeoff as follows. We allow every device to be involved in only one migration operation at a time. Thus, before issuing a migration request, the Migrator performs admission control by allowing requests only if the source and target device are both available. If they are not, the request is re-queued and it moves onto the next request. Further, the Migrator controls its migration-related resource consumption by decomposing an extent into smaller transfer units and *pacing* the transfer requests to match the minimum of the available or the desired I/O rate. Further if the migration is being performed to relieve throttling, once a transfer unit is migrated, any foreground I/O requests to it are handed by the destination array. Note that because of this pacing not all planned migrations may be completed before the next epoch. In such cases, the migration queue is flushed, and requests resulting from the new epoch's computation are queued. We further optimize by retaining the old location of the extent if it is already in the right tier during the consolidation step. Finally, we could potentially incorporate other

optimizations [AHH+01, DGJ+05, VKUR10, ZCD+10] such as multiple locations for the same extent [VKUR10], and proactive migrations [ZCD+10].

4.4.5 EDT Summary

The increasing availability of solid-state drives has ushered in a new era of multi-tiered primary storage systems. With EDT, we have formalized the *configuration* and *dynamic tier management* problems and have systematically explored the design choices available when building such systems. We presented the design, implementation, and evaluation of EDT's Configuration Adviser (EDT-CA) and Dynamic Tier Manager (EDT-DTM). EDT lowers capital cost by configuring less expensive tiered storage and operating costs by dynamically optimizing power consumption via consolidation whenever feasible. We also demonstrated that EDT is successfully able to address the data migration overheads of dynamic tiering and respond rapidly and effectively to unexpected changes in the workload.

Experimental results show EDT has significant benefit. Evaluation performed using both a production workload and industry-standard synthetic workload revealed that multi-tier systems using EDT have a device mix that saves between 5% to 45% in cost, consume up to 54% less peak power, and an additional 15-30% lower dynamic power (instantaneous power averaged over time), at a better or comparable performance compared to a homogeneous SAS storage system. Experimental results also demonstrated that EDT is superior to simpler alternatives for extent-based tiering, providing lower cost and better performance, and consuming similar or lesser power.

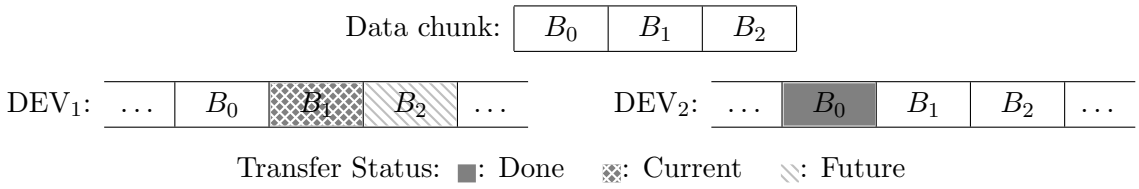


Figure 4.8: Possible cases when migrating data

4.5 Lessons Learned from Implementing Storage Extensions

In this section, we discuss the particularly challenging aspects of implementation storage extensions and what we have learned thus far.

4.5.1 Correctly Migrating Data

Moving data between devices is a core part of all our extensions. EXCES moves popular data to an ECDto save power, BORG re-arranges the data layout to optimize for sequential access, and EDTmoves data among different tiers to improve both performance and save power. But, doing this correctly and efficiently is not straight forward. In an optimistic case we migrate a piece of data and no I/Os are issued to it while this migration is ongoing. However, in reality this not always the case.

When and I/O comes to a chunk of data that is currently being migrated we consider three cases, as illustrated in Figure 4.8. Let DEV$_1$ and DEV$_2$ be the source and destination devices respectively. First, the data involved has been migrated then the I/O needs to be issue to DEV$_2$. Second, the data is currently being migrated, here we must for the I/O to complete and then send it to DEV$_2$. And third, the data is yet to be migrated, in which case we send it to DEV$_1$.

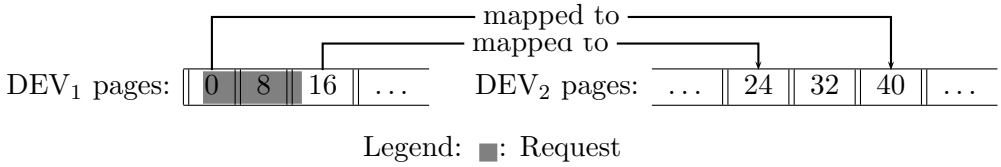


Figure 4.9: Alignment problem example

4.5.2 Indirection Implementation Issues

All the storage systems we have built so far group data in some unit. While this optimization allows us to cut down on metadata memory requirement, it complicates the implementation of the I/O indirection component. Since I/Os may be issued at unaligned 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. We address this issue via I/O request splitting and page-wise indirection.

Figure 4.9 shows an example of the alignment problem that the indirector must handle. Notice that two pages from DEV_1 are mapped to DEV_2. The first page on the disk that starts at block 0, is mapped to the fifth page on DEV_2 that starts from block 40. Also, the third page in the DEV_1 (starting at block 16), is mapped to the fourth page of DEV_2, starting at block 24. The second page in DEV_1 is not mapped to DEV_2 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. To address I/O splitting, we create one new request per page. 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.

4.5.3 Maintaining a Persistent Indirection Map

Since the systems that we propose modify data location on disk, we require some structure that allows us to mapping particular data to a device. This mapping requires strong consistency guarantees. We must ensure that reads are always up-to-date versions of data, including after a clean shutdown or a system crash.

In our system, map entries on-disk are updated (along with their in-memory version) each time the data is migrated or when a new map entry is added. But we have identified at least two problems with this approach. First, a write barrier is used to ensure that the on disk map is updated before additional entries are modified, this substantially reduces the performance. Second, and perhaps more importantly, sometimes we need to atomically write multiple data structures. For instance, a map update actually requires two writes; one for the free space bitmap, and other for the updated entry. Coordinating this two operations further degrades performance and complicates the indirection map update process. We believe that a generic mechanism that provides atomically updateable and durable data structures is possible. Such a mechanism should simplify the development of these extensions.

4.5.4 Programming Inside the OS Kernel

We implemented both BORG and EXCES as a Linux kernel module that can be dynamically inserted and removed without any changes to the kernel source. In the case of EDT, we chose to implement the intelligence of the system outside the kernel. However, we still relied on a kernel component that exported a virtual block

device and fowarded all I/Os to the user level component. Since the block layer interface of the Linux kernel is very stable, these systems can run "out of the box" on the latest 2.6 series kernels. The current implementation utilize native kernel data structures such as *radix trees* and *red-black trees* which are very likely to be retained in the future kernel versions.

Writing code inside the kernel is no easy task. It requires a substantial amount of skill and deep understanding of the workings the operating system as a whole. Additionally the kernel is a fragile programming environment, where a simple bug or program error can easily corrupt the entire system. Moreover, in many cases the documentation was non-existent and we would have to spend countless hours browsing through code in search of possible explanations of the observed behaviour.

One particular case was an error occurring while unloading the kernel module. While removal/unload we must ensure that the system is restored to consistent and usable state and foreground I/O handled correctly. For instance, EXCES must flush on-ECD dirty blocks to their original positions on disk. Initially we issued a write operation for every dirty block and invoked the module unload routine. But this naive approach had a problem and we were causing a system crash. Upon completion I/Os need to invoke part of the extension code. But, this code had just been removed from the running OS kernel thus the I/O interruption handler was trying to invoke non-existent code. We solved this problem by waiting for all extension related pending I/Os to complete and then we proceed with the module unload operation. This way we ensure that there are no in-flight extension I/Os by the time the extension is removed from the kernel. The solution seems obvious but finding the source of the problem was the hard part.

Apart from issues like this, we also notice that there was a substantial overlap in parts of the core functionality of the extensions. Code for inderecting I/Os,

migrating data, among others was largely the same. This simplified the development but also lead us to believe that a more efficient path is possible, where we have the common part of extensions grouped into a stable and easy to use library. We look into this in the following chapter.

4.6 Summary

In this chapter we argued for the block layer as an adequeate place to develop storage system extensions namely: a simple interface, file system independence, control of storage resources, among others. To illustrate that novel improvements that can be done at the block layer we presented the design and implementation of three extensions. In all cases the extensions provided quantifiable benefits when compared to the current practices. But, more important for our study were the lessons taken for the development experience. We learned that developing such extensions is no easy task. However, substantial part of the development process could be simplified provided developer abstractions for common functionality in storage extensions (e.g. consistently storing and retrieving extension metadata). In the next chapter we present an intrastructure with such abstractions that help relieve part of the burden of the storage extension developer.

CHAPTER 5

BUILDING AND STACKING STORAGE EXTENSIONS

Recent years has seen a significant amount of innovation in self-management extensions to storage systems [FMK+07, GPK+07, LCSZ04, MAC+08, Nar08, NDT+08, SWS05, ZLP08]. We can add to this list the self-managed block layer storage extensions we discussed in the previous chapter, namely BORG, EXCES, and EDT. However, wide-spread adoption of this innovation has been slow. Based on this experiences, we have identified two broad factors holding back large-scale adoption. First, developing storage extensions is a cumbersome and time consuming process. As we learned in the previous chapter, the operating system, the typical development environment for self-management extensions, is a complex building block that makes extensions extremely difficult to develop and validate. Adding to this are a host of scenarios and corner that developers need to take into account to guarantee consistency and recoverability. Second, reasoning about how deployed extensions affect the stored data and access it is an undeveloped science; consequently, administrators find themselves ill-equipped to make sound deployment decisions and choose "not to deploy what they don't understand".

The Active Block Layer Extensions (ABLE) project develops a theory and a systems infrastructure to address the two challenges outlined above. ABLE supports self-management extensions that built at the *block layer* that export a logical block interface to storage clients (e.g. file systems) for accessing an underlying local or remote block device. Block layer extensions automatically inherit several desirable properties such as access to both process and device context of I/O operations, file-system independence and accessibility and among others which we examine in detail in § 5.2.

The ABLE project makes two contributions. The first is an an evolvable block-layer software infrastructure that implements a suite of block layer primitive functionality, commonly used by self-management extensions, as an in-kernel library. These primitives raise the level of abstraction for developing self-management extensions, helping developers build robust self-management extensions faster. The second contribution is a theory of block layer extensions that provides a logic framework for understanding how storage extensions affect the data path. This theory enables modeling the behavior of individual extensions and analyzing the influence of extension aggregates on data and data accesses. In this work, we assume that each extension ensures block consistency for data that it handles. Analysis tools based on this theory can help administrators make sound deployment decisions that accurately reflect high-level policies when composing a storage system using self-management extensions as building blocks.

In the rest of the section, we first present the motivation behind ABLE's design decisions (§ 5.1) and elaborate on the key elements of the architecture and design (§ 5.2). We then develop a novel theory to reason stacking of block layer extensions (§ 5.3) and evaluate its use in practise (§ 5.4). We quantify the reduction in development complexity when using ABLE (§ 5.5).

5.1 Infrastructure Requirements

This section examines the requirements for an storage self-management infrastructure from the standpoint of both developers and administrators of storage systems.

5.1.1 Requirements of the Developer

Storage systems extensions typically extend storage stack implementations for minimizing overhead. However, the storage stacks are cumbersome to extend. The de-

velopment abstraction is low and an overwhelming amount of detail of a pre-existing codebase must be mastered prior to development. Moreover, the highly concurrent environment with multiple priority levels of process and interrupt contexts make thread-safe data manipulation and blocking operations complex [ABB+86]. These factors make the development experience poor and lower the confidence on the correctness of developed extensions.

Our experiences with storage extensions revealed (§ 4.5) that a software infrastructure within the storage stack can factor out a substantial fraction of non-functional extension complexity [Dav93] into a common codebase or library. These non-functional components require in-depth understanding of I/O handling inside the kernel and are often the main source of bugs during development. Non-functional components that are commonly desirable across storage extension development processes include support for: *(i)* interposition on I/O operations both in the *issue* and *completion* paths, *(ii)* monitoring system state, *(iii)* common storage operations of data manipulation, I/O manipulation, and data movement, *(iv)* exclusive access to system resources (e.g., memory and disk space), and *(v)* simple and dynamic management of extensions in an active storage stack.

5.1.2 Requirements of the Administrator

Managing self-management extensions has the potential to substantially increase the *incidental complexity* [CHIK03] involved in system administration. Since storage self-management extensions impact the access path to persistent data, a system administrator would need to evaluate the impact of a candidate extension on I/Os issued to each data volume. When deploying multiple extensions, she must be able to reason about the impact of each extension, on each managed volume. However, no formal theory exists that would help confidently reason about how multiple ex-

tensions inter-operate. In the absence of precise knowledge about the operation of a each extension, the administrator must either employ anecdotal evidence or use trial-and-error. Since the number of possible ways to stack extensions grows exponentially with the number of extensions and each combination may have a completely different behavior (as we shall illustrate in § 5.3.1), these combinations must be analyzed individually. We identify the following capabilities for a storage system infrastructure to simplify administration: *(i)* clearly describe the impact of an extension to target data volumes, *(ii)* automate and/or provide sound hints for extension deployment that reflect admin-specified system policies, and *(iii)* monitor resource usage characteristics of live extensions.

5.2 ABLE Architecture and Design

The requirements laid out above motivate the architecture and design of ABLE.

5.2.1 ABLE Architecture and Design

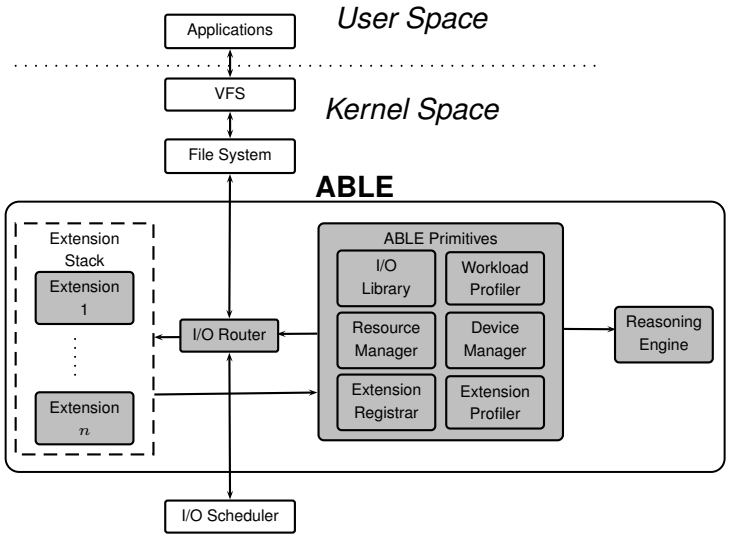


Figure 5.1: ABLE architecture. Arrows depict control flow.

Figure 5.1 presents the architecture of ABLE, designed as an extension of the block layer in the storage stack. ABLE's interaction with other layers is handled by the *I/O Router*, which directs each block I/O request through individual extensions in the *Extension Stack*, based on the attributes of the I/O request and the configuration of the extensions. Each extension processes I/Os routed to it; that may include invoking *ABLE Primitives*. The ABLE primitives may interact with the *Reasoning Engine* to manage the stack correctly. Next we present a detailed description of each component.

Extension Stack Comprises currently deployed self-management extensions. Extensions are stacked according to rules that we describe in § 5.3 by the reasoning engine. Each extension has an input and an output *domain* of operation configured by the system administrator; the *input domain* specifies the data volume(s) that the extension is configured to operate upon, while the *output domain* specifies the data volume(s) to which these block I/O requests are directed by the extension.

I/O Router Manages ABLE's interaction with other layers in the storage stack as well as internal routing of I/O requests and I/O completion events between extensions. For each I/O request made by the layer above, the I/O router checks if its destination matches the input domain of individual extensions in the stack in top-down order. If yes, the appropriate I/O handler for the extension is invoked. After an extension finishes processing an I/O request, it registers an I/O completion handler and returns control to the I/O router which decides the next extension to route the I/O to. Upon I/O completion, the I/O router invokes the I/O completion handlers of individual extensions in the reverse order of their invocation during the forward path, finally invoking the the handler of the layer above. Additional I/Os may be issued by an extension; in such cases, the I/O router will route such requests only via the ones stacked below the generating extension.

Sub-Component	Interface
I/O Library	`clone_bio, submit_bio, submit_bio_nonblock, copy_blocks, gather_blocks, scatter_blocks`
Resource Manager	`persistent_malloc/free, persistent_restore, get_resource_usage`
Workload Profiler	`set/get_request_information, block_accessed, get_popular_blocks`
Device Manager	`get_device_characteristics, get_device_state, alloc/free_simulated_disk`
Extension Registrar	`[un]register_extension, register_extension_at, query_register_extension`
Extension Profiler	`total/avg_extension_processing_time, total/avg_generated_IOs`

Table 5.1: ABLE primitives (not a complete list).

Reasoning Engine In typical usage, the system administrator will use the ABLE infrastructure to deploy multiple self-management extensions that may be written by independent developers. Apart from choosing which extensions to deploy, she will configure their input and output domains and specify the system's self-management policies. This component provides a tool for automatically analyzing and composing extension stacks to reflect administrator-defined self-management policies, based on the properties and operational domains of individual extensions (formalized in § 5.3).

5.2.2 The ABLE Primitives

The ABLE primitives extend storage stacks to *(i)* simplify development of robust self-management extensions and *(ii)* aid administrators in deploying extensions to best reflect defined system priorities and policies. Table 5.1 list the key ABLE primitives, categorized based on its six sub-components. These primitives provide core support for extension development and administration.

The *I/O library* provides support for extension development with a set of functions for commonly performed block I/O operations. These include primitives

to copy_blocks using source and destination locations specified by the caller, submit_bio_nonblock so that the caller may issue an I/O operation from an interrupt context, gather_blocks that are spread across a volume to a contiguous defined area *efficiently* given a map of source and target block addresses, etc. Some of the above primitives generate new I/O requests (e.g., copy_blocks), which are handled by the I/O Router and considered as issued by the caller extension.

The *Resource Manager* provides primitives to allocate resources from the system. The persistent_malloc and persistent_free primitives allocate and free respectively a contiguous piece of memory and automatically maintains it persistent by mapping such memory to persistent disk locations. To restore an in-memory data structure (due to a system restart, crash, or power-failure), the caller invokes persistent_restore. The design and techniques employed to handle memory persistence are discussed in Chapter 7.

The *Workload Profiler* manages information about the requests being produced by the system and the extensions. This data can be utilized by individual extensions and other components for decision making. For instance, a caller can obtain the top-K frequently accessed blocks using an appropriately parameterized get_popular_blocks invocation. The *Device Manager* controls allocation of storage resources so extensions can request and independently manage storage space, both *real* and *simulated*. A developer can use alloc_simulated_disk to allocate a simulated volume to test her extension; such simulated volumes can be fully-simulated (for quickly exploring various code-paths) or memory-simulated (for data consistency testing). Extensions can also query the state and current usage of both physical and virtual devices using get_device_state. Extensions may also register handlers with the *Device Manager* to be notified of important changes in device state (e.g. plug, unplug). In a similar way, the *Extension Profiler* keeps track of

extension resource usage and other statistics, which can be used by the *Reasoning Engine* to infer changes in the system and take necessary action based on system goals, and can also be used directly for guiding manual administration.

The *Extension Registrar* handles extension registration and deregistration. In a request for extension insertion to the registrar, the administrator may either choose a specific position in the stack to place it (using `register_extension_at`) or allow the position to be determined by ABLE (using `register_extension`). In the latter case, *Reasoning Engine* primitives are invoked to determine its position in the stack. Subsequent actions ensure that the new extension is successfully incorporated to the system, including initializing extension variables, loading of persistent metadata, and marking the extension as usable.

5.2.3 ABLE and File Systems

The ABLE infrastructure can also aid file system layer optimizations in a straightforward manner. Particularly, file system extensions can readily use the I/O library, workload profiler, and resource manager sub-components of the ABLE primitives to simplify their development. Requests generated due to invocations of the I/O library API are handled by the I/O router in the same as I/Os generated directly within the file system. Further, complex file system optimizations can isolate block-level concerns within an ABLE extension. For instance, recent abstractions that aid reliability and performance in file systems (e.g., I/O shepherding [GPK+07], patches [FMK+07]), can largely be encapsulated as ABLE extensions, creating a generalized capability inside the OS that other file system implementations as well as other storage clients (e.g. virtual memory) can use.

5.3 A Theory of Block Layer Extensions

Our primary goal for developing a theory of block layer extensions was for use in administrative tasks. Consequently, we were primarily interested in reasoning about the influence of storage extensions on *data volumes*, the administrative units of storage management, rather than storage devices. Further, we simplify the reasoning framework to focus on steady-state system behavior and thus exclude considerations of extension operation during their initialization and finalization phases.

5.3.1 An Illustrative Example

To illustrate the importance of extension stacking order, let us consider two real-world self-management extensions. Extension $\mathcal{IS}$ improves the reliability of data stored and accessed on a local disk volume D via an I/O Shepherding [GPK+07] optimization. Extension $\mathcal{EX}$ uses a volume F on a locally-attached, low power flash device to prefetch, cache, and buffer data stored on D so the local disk may be powered down whenever possible to save energy [UGB+08]. Both extensions generate additional I/O operations besides the application-generated I/Os to accomplish their respective tasks; $\mathcal{IS}$, for instance, retries failed I/O operations, while $\mathcal{EX}$ moves data between D and F. If both extensions are chosen to be deployed, two possible stacking choices exist (showed in Figure 5.2): (i) $\mathcal{IS}$ above $\mathcal{EX}$, or (ii) $\mathcal{EX}$ above $\mathcal{IS}$. With the first option, substantial additional traffic, including duplicate "Mirror" writes and "Retry" reads from $\mathcal{IS}$ is handled by $\mathcal{EX}$, favoring reliability over energy saving. Alternatively, placing $\mathcal{EX}$ above $\mathcal{IS}$, $\mathcal{IS}$ reliability mechanisms are not invoked when applications access data cached in F, thus compromising reliability for energy saving.

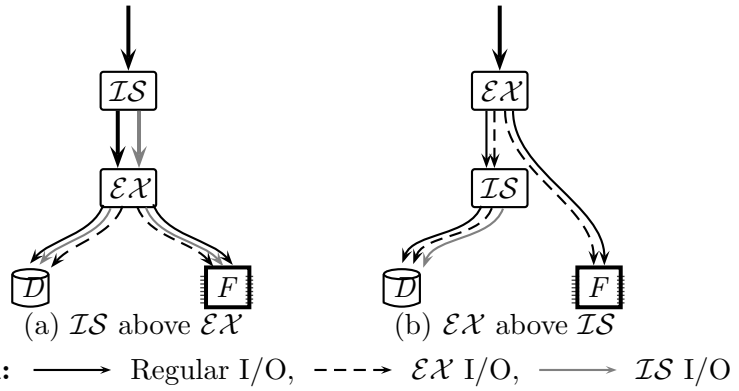


Figure 5.2: Stacking options for extensions $\mathcal{IS}$ and $\mathcal{EX}$. Lines represent I/O request flow, thicker lines indicate greater I/O flow, gray and dashed lines indicate requests created by $\mathcal{IS}$ and $\mathcal{EX}$ respectively. D is the disk drive and F is a flash device.

An administrator, typically not versed with the intricacies of how the extensions operate, would be ill-equipped to perform the above analysis to lead to a correct deployment choice. However, what the administrator can determine confidently is system priorities, in terms of the relative importance of data reliability over power-savings at any given time and for a given data volume.

5.3.2 Concepts and Definitions

We begin by formalizing the concept of extension goal.

Definition 5.3.1 *The* **goal** *of an extension is the system metric that the extension improves.*

An extension goal can be any of the system metrics of interest such as performance, reliability, energy savings, security, etc. For instance, the goal of the $\mathcal{IS}$ and $\mathcal{EX}$q extensions introduced previously are reliability and energy savings respectively. These goals are specified by the developer of the extension, chosen from a standardized set. Likewise, the system administrator can maintain a prioritized list of *system* goals for each data volume.

The domain of operation for an extension is the set of data volumes that may be affected by the extension during its steady-state operation. More formally, we define the concepts of *input* and *output* domains.

Definition 5.3.2 *The* **input domain** *of an extension is the set of data volumes on which the extension operates in its steady-state; its* **output domain** *is the set of volumes to which I/O operations, both incoming and self-generated, may be directed by the extension in steady-state.*

Thus, an extension interposes on each I/O operation targeted to data volumes in its input domain, and directs I/Os to data volumes in its output domain. For instance, the input and output domains for the $\mathcal{EX}$ extension described previously would be $\{D\}$ and $\{D, F\}$ respectively. We anticipate that the administrator would define the input and output domains of any extensions she may wish to deploy.

Extensions may alter the I/O traffic in distinct ways; some can be non-intrusive, observing requests to collect statistics, while others may divert the I/O stream to a different volume. Data accesses to any data volume depends on the influences of various extensions on the I/O traffic to that volume. We model an extension's influence on I/O traffic with three categories of extension properties: *accessors, mutators*, and *generators*.

Definition 5.3.3 *Extensions with the* **accessor** *property only observe the I/O stream and do not modify the requests in any way (e.g., an I/O profiling extension).*

Definition 5.3.4 Mutator *extensions may modify any part of the individual requests within the I/O stream, including the data payload, or the target location of the I/O request (e.g., a block encryption extension).*

Definition 5.3.5 Generator *extensions may create additional I/O requests, triggered by workload, system changes or failures (e.g., a* read-after-write *extension).*

Extensions will have one or more of these properties. Using such properties we can further characterize extension behavior by specifying the output of an extension. Let K be an extension, O_K the output domain of K. Let $\mathbb{A}_K$, $\mathbb{M}_K$, and $\mathbb{G}_K$ be the subsets of the output domain O_K that were exclusively accessed (part of the input not modified in any way), mutated (part of the input modified), and generated (not part of the input, created new output) by the extension respectively. More formally we have,

$$
\begin{aligned}
\mathbb{A}_K \neq \varnothing &\Rightarrow \text{K has the accessor property} \\
\mathbb{A}_K &\subseteq O_K \\
\mathbb{M}_K \neq \varnothing &\Rightarrow \text{K has the mutator property} \\
\mathbb{M}_K &\subseteq O_K \\
\mathbb{G}_K \neq \varnothing &\Rightarrow \text{K has the generator property} \\
\mathbb{G}_K &\subseteq O_K \\
\mathbb{A}_K \cap \mathbb{M}_K \cap \mathbb{G}_K &= \varnothing \\
\mathbb{A}_K \cup \mathbb{M}_K \cup \mathbb{G}_K &\subseteq O_K
\end{aligned}
$$

The *block consistency contract* requires that each block read contains the exact same data that was last written to it. This contract dictates safety in implementation for extension property class. An accessor is safe by definition. A mutator must ensure that each mutation of block written is reversible; it must restore the original contents upon a read operation. A generator must ensure that any additional write requests it issues do not overwrite blocks owned by other entities (e.g., file systems, other ABLE extensions). While the above properties are useful for formalizing extension behavior, some extensions may possess more than one property at once.

For instance, a RAID-1 extension which mirrors each write request on two volumes and redirects read traffic depending on head position in each drive has both generator and mutator properties.

Two final concepts, *effectiveness* and *dominance*, formalize the relative influence of individual extensions on data volumes.

Definition 5.3.6 *An extension within a given stack of extensions is said to be* **effective** *over a specific data volume if its operational goal is met for all I/Os ultimately directed to the data volume.*

Definition 5.3.7 *An extension within a given stack of extensions is said to be* **dominant** *over a specific data volume if it is the final handler of all I/Os ultimately directed to the data volume.*

Both effectiveness and dominance address I/Os to the data volume both from the application and due to other extensions in the stack, but excluding application I/Os originally directed to the data volume but that have been redirected by other extensions to other data volume(s). We observe that an extension E that is *dominant* over a data volume V is also *effective* over V, but the converse is not necessarily true. For instance, in Figure 5.2(a), $\mathcal{IS}$ is effective over volumes D and F but is not dominant over either, while $\mathcal{EX}$ is dominant (and also effective) over both volumes D and F.

Given a specific extension stack configuration with the domains and operational goals of each extension (e.g., reliability, performance, etc.), we develop a theory that comprehensively establishes *dominance* of extensions over the managed data volumes. While *effectiveness* of extensions is also important to reason about to fully understand the influence of extensions on volumes managed, our theory is incapable of reasoning about effectiveness comprehensively. The partial reasoning

about effectiveness that we do explore serves to articulate the limitations of our theory.

	$O_A \cap I_B = \varnothing$	$O_A \subseteq I_B$	$O_A \not\subseteq I_B \wedge$ $O_A \cap I_B \neq \varnothing$
$I_A \cap I_B = \varnothing$	A is dominant $\forall x \in I_A$ iff. $O_B \cap I_A = \varnothing$; B is dominant $\forall x \in I_B$	B is dominant $\forall x \in I_A \cup I_B$	B is dominant $\forall x$ s.t. $x \in I_B \vee f_A(x) \in I_B$; A is dominant $\forall x$ s.t. $f_A(x) \notin I_B$ iff. $O_B \cap I_A = \varnothing$
$I_A \supseteq I_B$	A is dominant $\forall x \in I_A$	B is dominant $\forall x \in I_A$	B is dominant $\forall x$ s.t. $f_A(x) \in I_B$; A is dominant $\forall x$ s.t. $f_A(x) \notin I_B$ iff. $O_B \cap I_A = \varnothing$
$I_A \not\supseteq I_B \wedge$ $I_A \cap I_B \neq \varnothing$	A is dominant $\forall x \in I_A$ iff $O_B \cap I_A = \varnothing$; B dominates $\forall x \in I_B - I_A$	B is dominant $\forall x \in I_A \cup I_B$	B is dominant $\forall x$ s.t. $x \in I_B - I_A \vee f_A(x) \in I_B$; A is dominant $\forall x$ s.t. $f_A(x) \notin I_B$ iff. $O_B \cap I_A = \varnothing$

Figure 5.3: Extension Stacking Rules Grid. These rules assume there are only two extensions in the stack, A and B, with A stacked over B. x is a block (represented by its address) within some data volume managed by A and B. I_K and O_K represent the input and output domains of an extension K respectively. $f_K(x)$ is the resultant request after x is processed by extension K.

5.3.3 Generalized Extension Stacking

The generalized stacking rules are the cornerstone of the ABLE theory. These rules establish dominance relationships between extensions and data volumes and can be applied to a collection of extensions, regardless of their properties or goals. ABLE uses these rules to generate automatic hints for the administrator on stack configurations that reflect the prioritized list of goals for each data volume. These rules can also be used by an administrator to reason about the overall behavior of alternative extension stack configurations.

We simplify the rule-set by considering a pair of extensions A and B, where A is stacked above B, and establish dominance for their target data volumes. This simplification is made without loss of generality because larger stacks can be reasoned piece-wise using the simplified rule-set by partitioning the stack recursively into two groups, one with a single extension at one end of the stack (say at the top) and the

rest grouped into a single unit. Dominance information for the single extension at the top can be immediately established, while such information for each extension in the group of extensions can be obtained recursively.

Figure 5.3 presents nine possible cases based on the relationship between the input and output domains of extensions A and B, which together comprise the *stacking rules grid*. The purpose of the rules is to enable reasoning about the impact of the extensions on a request (represented by the variable x to denote a logical block) to data volumes. Rules grid element *(i,j)* represents the rule defined at row i and column j. Owing to space constraints, we provide the formal proofs for each of these rules in an online document [GR] and only briefly explore the insights behind some of the rules next.

When examined columnwise, we observe that if the output domain of A is disjoint of the input domain of B, dominance is primarily determined by the relationship between the input domains, with a additional constraints in some cases as described below. In case the extensions operate on disjoint input domains, i.e., $(1, 1)$, B is dominant over its input volume. However, A is dominant over its input volume iff. the output domain of B (O_B) and the input domain of A (O_A) are disjoint as well; otherwise, I/O requests redirected by B towards I_A are not handled by A thereby compromising the dominance of A over I_A. In the next column, when the output domain of A is a subset of the input domain of B, i.e., $(2, *)$, regardless of extension A's manipulation of the request stream, all resultant requests are squarely directed within the input domain of B; extension B is thus dominant over the input domain of both extensions. For the last column, i.e., $(3, *)$, since a subset of the output domain of A is not in the input domain of B, not all requests processed by A will be observed by B and thus, a single argument cannot be made about the entire output domain A. If more information about the function f_A that maps elements (data

volumes) in the input domain of A to its output domain is available, we can argue that B is dominant for those elements in the input domain of A that are mapped to the input domain of B, while A is dominant for the rest of its input domain iff. the output domain of B is disjoint of the input domain of A.

5.3.4 On Effectiveness and Theory Limitations

While the stacking rules described above do not reason about effectiveness, we make the following additional observations to address some scenarios where additional knowledge can be inferred. Given extensions A and B, let us assume that A is stacked above B.

Theorem 1 *If* $\mathbb{A}_B \subseteq O_A$ *and* $(I_A \cup O_A) \cap (\mathbb{G}_B \cup \mathbb{M}_B) = \varnothing$*, then* A *is effective.*

The above theorem states that A is *effective* over its input domain if B does not modify requests handled by A or generate additional requests to A's target data volume. From an abstract point of view this theorem is simply stating that if B does not modify in any way the data domains of extension A, then A is effective. We can observe that a particular instance of the above theorem occurs when B only has the *accessor* property in which case A is always effective. This can be formalized as follows:

Corollary 1 *If* $\mathbb{G}_B \cup \mathbb{M}_B = \varnothing$*, then* B *only has the accessor property and* A *is effective.*

When B is a *generator* or *mutator*, A's effectiveness is dependent on the semantics of extension B. While B may further modify the request stream handled by A, it may do so in a complementary fashion and thereby retain the effectiveness of A for its target data volumes. For instance, in the example presented in Figure 5.2(a),

although rules grid element $(2, 2)$ applies, $\mathcal{IS}$ is still effective over volumes D and F when stacked over a extension $\mathcal{EX}$, because $\mathcal{EX}$'s mutation/generation of requests are complementary to those of $\mathcal{IS}$. This reasoning for the effectiveness of an extension A when an underlying extension B is a generator or mutator finely articulates the limitations of our theory.

Finally, there is one additional case where we can argue about an extension being ineffective. Again let us assume that A is stacked above B. If the input of B is nullified due to the actions of A, then B is ineffective under this stacking order. This is formally stated in theorem bellow.

Theorem 2 *If $I_A \supseteq I_B$ and $O_A \cap I_B = \varnothing$, then B is ineffective.*

5.3.5 An Illustration of Stacking Rules

We provide more insight into the practical implications of the stacking rules using six ABLE extensions considered for deployment on a target system. The target system has two local persistent storage devices, a disk drive (D), and a flash-based solid-state drive (F), and a remote iSCSI target used as a backup drive (D_r). We also consider the system memory (M) as a storage device, since it is used as a first-class resource by one of the extensions described below.

Block Versioning [MG03, FB04] ($\mathcal{BV}$): Provides data backup by continuously maintaining all versions of blocks written to local devices D and F, on the remote iSCSI target D_r for reliability.

Data Deduplication [QD02] ($\mathcal{DD}$): Deduplicates data blocks on the remote drive D to reduce used space.

I/O Off-Loading (extends write off-loading [Nar08] to reads as well) ($\mathcal{IO}$): Temporarily transfers all I/O activity on D to a remote drive D_r (serving as mirror or versioning-block store) so the local disk can be turned off to save energy.

Managed Flash Technology [Eas08] ($\mathcal{MF}$): Implements a log-structured block-layer for the local flash device F to address the performance of random writes..

I/O Shepherd [GPK+07] ($\mathcal{IS}$): Improves reliability of data accessed on local devices D and F by using a clearly defined set of policies to deal with different types of I/O errors. Examples include retrying failed I/O operations and automatic block mirroring.

Local Cache ($\mathcal{LC}$): Uses the memory device M to cache frequently accessed blocks from the slow local disk drive D to improve system performance.

Most of the extensions above are hybrid extensions that satisfy both *mutator* and *generator* properties, except for $\mathcal{BV}$, which is only a generator, and $\mathcal{LC}$, which is only a mutator. Further, accurately identifying properties requires careful analysis; for instance, extension $\mathcal{DD}$ may appear to be a pure mutator (it modifies block addresses of I/O requests), but closer examination would reveal that it generates additional I/O operations to keep its metadata up-to-date. Properties of the extension are therefore best specified by its developer. Figure 5.4 presents the input and output domains of these sample extensions.

Ext / **Dom**	$\mathcal{BV}$	$\mathcal{DD}$	$\mathcal{IO}$	$\mathcal{MF}$	$\mathcal{IS}$	$\mathcal{LC}$
Input	D, F	D_r	D	F	D, F	D
Output	D, F, D_r	D_r	D_r	F	D, F	D, M

Figure 5.4: Domains definitions for sample extensions.

Using these extensions we exemplify the 9 rules in terms of input and output domains using in each case, a pair of extensions, as shown in Figure 5.5. Due to space

	$O_A \cap I_B = \varnothing$	$O_A \subseteq I_B$	$O_A \nsubseteq I_B \wedge$ $O_A \cap I_B \neq \varnothing$
$I_A \cap I_B = \varnothing$	$\frac{\mathcal{MF}}{\mathcal{LC}}$	$\frac{\mathcal{IO}}{\mathcal{DD}}$	$\frac{\mathcal{BV}}{\mathcal{DD}}$
$I_A \supseteq I_B$	$\frac{\mathcal{IO}}{\mathcal{LC}}$	$\frac{\mathcal{IS}}{\mathcal{BV}}$	$\frac{\mathcal{BV}}{\mathcal{IS}}$
$I_A \not\supseteq I_B \wedge$ $I_A \cap I_B \neq \varnothing$	$\frac{\mathcal{IO}}{\mathcal{IS}}$	$\frac{\mathcal{MF}}{\mathcal{IS}}$	$\frac{\mathcal{LC}}{\mathcal{BV}}$

Figure 5.5: Sample stack configurations illustrating the stacking rules grid elements depicted in Figure 5.3.

constraints, we only examine three stack configurations corresponding to elements (2,1), (3,2), and (1,3) respectively of the stacking rules grid.

(2,1) $\mathcal{IO}$ is stacked above $\mathcal{LC}$. Per the rules grid, $\mathcal{IO}$ is dominant; its operation is preserved. This is easily confirmed by noting that since the I/O offloading extension intercepts all I/O requests to D the block layer first, it diverts all the accesses to D instead to the remote device D_r; the caching extension, does not observe any I/O traffic at all, and is thus not dominant.

(3,2) $\mathcal{MF}$ is stacked above $\mathcal{IS}$. Per the rules grid, $\mathcal{IS}$ is dominant. We observe first that read operations pass through the Managed Flash extension unaffected. Second, while write operations are possibly redirected to a different location on F by $\mathcal{MF}$, these are still intercepted by the I/O shepherding extension which applies its reliability policies. On the other hand, any additional write I/O operations generated by $\mathcal{IS}$ are not observed by $\mathcal{MF}$ thus compromising $\mathcal{MF}$'s dominance.

(1,3) $\mathcal{BV}$ is stacked above $\mathcal{DD}$. Per the rules grid, $\mathcal{DD}$ is dominant for all additional write operations generated by the block versioning extension, i.e., those directed to the remote drive D_r, the input domain of $\mathcal{DD}$. This is easily confirmed since each of the write I/O generated by $\mathcal{BV}$ are processed by $\mathcal{DD}$, which is lower in the stack.

5.4 Putting Theory into Practice

Deploying self-management storage extensions raises several questions conclusively: *(i)* how does a deployed extension X affect data volume V?, *(ii)* how will extensions interact when deployed?, and *(iii)* how can extensions be configured so per-data-volume goals are met? In the absence of in-depth knowledge about the extensions and the inherent complexity of the required reasoning, administrators would much rather prefer such questions to be answered comprehensively by a tool that provides deployment decisions.

The Reasoning Engine (RE) component of ABLE is based on the theoretical foundations developed in § 5.3 and implements the logic framework to formally reason about these questions and thus compose storage stacks correctly. The RE is responsible for adding extensions to the stack correctly and can be used in two modes: *automatic* and *querying*. In *automatic mode*, ABLE automatically suggests the stack configuration that maximizes the dominance of extensions that meet goals with higher priority, with prioritized system goals defined by the administrator. This process and dominance maximization are further elaborated in § 5.4.4. In *querying mode*, the administrator can query the RE with a candidate extension being considered for deployment. The RE will list all possible positions in the current stack, where the extension could be added and identify the goals that are accomplished in each scenario, as well as a list of which extensions are dominant for each data volume. mechanisms) in a practical administration scenario.

5.4.1 The Administration Scenario

Suppose an administrator manages a system with a locally-attached data volume (D) and a remote iSCSI target volume (D_r) with large storage capacity and inbuilt

redundancy. The system is set up with a continuous block versioning mechanism that for each block written to D, synchronously generates an additional write operation to the remote volume D_r. However, currently the administrator must address two immediate concerns with the system's operation: free space on the target volume D_r is being consumed at a much faster rate than anticipated, and power consumption (and resulting heat) of the system is consistently exceeding acceptable levels. She considers dealing with these issues by deploying two ABLE-based self-management extensions. First is a *data deduplication* ($\mathcal{DD}$) extension for D_r to reduce its space utilization. Second, is an *I/O offloading* ($\mathcal{IO}$) extension that periodically offloads I/O operations entirely to D_r so that the local disk D can be powered-down for long periods to reduce the system's power consumption.

5.4.2 Configuring Extensions and System Goals

The goals of the two extensions, $\mathcal{DD}$ and $\mathcal{IO}$, have been specified by their respective developers as *space utilization* and *energy savings* respectively. The first task for the administrator is to define the operational *domains* for $\mathcal{DD}$ and $\mathcal{IO}$. This is a straightforward task since she is very clear about the intended use of the extensions. She sets up the I/O domain configurations as seen in Table 5.2.

Extension	Input Domain	Output Domain
$\mathcal{DD}$	D_r	D_r
$\mathcal{IO}$	D	D_r

Table 5.2: I/O domain configurations for $\mathcal{DD}$ and $\mathcal{IO}$.

Next, the administrator must define a prioritized list of system goals. She chooses to prioritize space utilization (on D_r) over power savings (for D), since the space utilization on D_r is a more urgent concern.

5.4.3 Identifying the Stacking Order

For simplicity, let us assume that extensions $\mathcal{DD}$ and $\mathcal{IO}$ must be stacked below the already active block versioning extension. Consequently, I/O traffic to both volumes (D and D_r) must be routed appropriately via the new extensions $\mathcal{DD}$ and $\mathcal{IO}$. To decide the stacking order for deploying $\mathcal{DD}$ and $\mathcal{IO}$, the administrator consults the ABLE's RE-based tool in *automatic mode* which recommends stacking $\mathcal{IO}$ above $\mathcal{DD}$, given the prioritized system goal of space utilization. It further informs that with the above stacking order, the space saving $\mathcal{DD}$ would be dominant over volume D_r. While this recommendation simplifies the task of the administrator, a curious administrator can additionally explore dominance for the alternative stacking option with the tool's *querying mode*.

To validate the correctness of the tool's recommendation, let us further analyze both stacking options — $\mathcal{DD}$ *above* $\mathcal{IO}$, and $\mathcal{IO}$ *above* $\mathcal{DD}$. Considering $\mathcal{DD}$ stacked above $\mathcal{IO}$, the $(1, 1)$ element of the rules grid (Figure 5.3) applies, indicating that while $\mathcal{IO}$ is dominant for D, $\mathcal{DD}$ is not dominant for D_r since $O_{IO} \cup O_{DD} \neq \varnothing$. To understand the above outcome, let us consider a sequence of two write I/O operations. The first is to the local volume D – let us name this request d_1. The second write request – named d_2^r – is to the remote volume D_r, but the content being written is a duplicate of d_1.

Figure 5.6(a) depicts how the first stacking option will handle this sequence of two writes. Request d_1 will only be processed by $\mathcal{IO}$, which will redirect it to the remote volume D_r – let us call this modified write d_1^r; in addition, $\mathcal{IO}$ will generate a meta-data write request to keep track of the dirty block (not up-to-date on D) in D_r to ensure consistency. On the other hand, d_2^r will only be processed by $\mathcal{DD}$ which will record its content-hash [QD02] and let it proceed to its destination D_r.

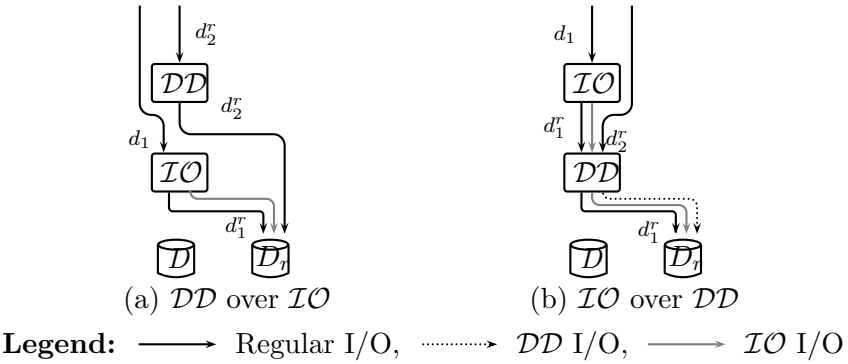


Figure 5.6: I/O handling with alternative stacking of extensions $\mathcal{IO}$ and $\mathcal{DD}$. Arrows represent I/O requests. d_1 and d_2^r are two writes with identical contents originally addressed to the local and remote volumes respectively.

Thus, with this stacking option, extensions observe distinct I/O requests and at a high level seem not affect each other, but $\mathcal{IO}$ does generate requests for D_r that are not handled by $\mathcal{DD}$, thus compromising the effectiveness of $\mathcal{DD}$.

Let us now examine the second stacking option, $\mathcal{IO}$ above $\mathcal{DD}$ (Figure 5.6(b)). This time we apply rule (1,2) from the stacking rules grid indicating that extension $\mathcal{DD}$ is dominant for both local (D) and remote volumes (D_r). As before, $\mathcal{IO}$ will redirect d_1 to the remote volume D_r, thus creating d_1^r. However, this time d_1^r will be processed by $\mathcal{DD}$ since it falls within its input domain. $\mathcal{DD}$, in turn, will record its content-hash before letting it proceed to the remote volume D_r. When the second request d_2^r arrives, only $\mathcal{DD}$ will process it based on its input domain specification. Based on d_2^r's content-hash, it is a write request with duplicate content. $\mathcal{DD}$ will not forward it to the target device, effectively deduplicating content, and creating free space. Instead it will generate a meta-data write request to update an entry in its persistent indirection map to denote that read requests to location d_2^r must henceforth be redirected to d_1^r.

Considering a contrary scenario, if the administrator had prioritized power reduction over space utilization, the recommendation of ABLE's RE would be the

alternative stacking order, $\mathcal{DD}$ above $\mathcal{IO}$, with $\mathcal{IO}$ being the dominant extension for D the local disk.

5.4.4 Automatic Extension Stack Composition

There are two alternative methods supported by the RE to automatically compose extension stacks in ABLE. The first enables deploying a new extension without modifying the order of the extensions that have already been deployed. An alternate use supports the composition of the stack from scratch to deploy several extensions at once.

In the first usage, given a stack with n extensions, ABLE considers all $n + 1$ possible positions for the new extension, resulting in $n + 1$ candidate stacks. It recursively partitioning each candidate stack as described in § 5.3.3 and applies stacking rules are applied to infer dominance of extensions on each data volume. From these candidate stacks, ABLE selects the one(s) that maximize the dominance of extensions that meet higher priority goals, before attempting to maximize the dominance of extensions that meet lower priority goals. *Maximizing dominance* of an extension requires maximizing the number of data volumes for which the extension is dominant. This metric is used with the assumption that lower-priority goals can be sacrificed in favor of higher-priority goals across all data volumes. This stacking approach can also be adapted to support independent goal priorities per data volume. If multiple such candidates are found, ABLE chooses the one with the largest number of lower priority goals being accomplished.

The above approach may not yield the best solution (in terms of accomplishing the desired goals) since a reduced solution space is considered. If we allow modifying the order of extensions currently in the stack, an optimal solution can be guaranteed. In this alternate use of RE, ABLE considers all $(n + 1)!$ permutations of the $n + 1$

extensions, calculating in each case the dominance of extensions per volume using the stacking rules, and selecting based on the dominance maximization criteria as used in the previous method.

5.5 Evaluating the ABLE Infrastructure

In this section, we evaluate the benefits of the ABLE infrastructure using several ABLE-based extensions. For each extension, we compare two variants, one implemented without using the ABLE primitives (called vanilla) and the other based on ABLE primitives.

5.5.1 Complexity Metrics

In each evaluation, we measure the difference in *lines of code* (LOC), and analyze the factors contributing to the difference. We also use the *McCabe cyclomatic complexity* (CC) [McC76], that measures the number of linearly-independent paths through a program. While the LOC and CC provide high-level comparative mechanisms, development of code inside the kernel is a substantially more intricate process. First, a line of code that requires consulting complex data structures and function definitions within several kernel source files can be arbitrarily more complex to develop than one that does not. Second, developing code that runs in interrupt context requires more consideration than code that does not. Finally, synchronization is inherently complex inside the kernel due to multiple levels of interrupt and user contexts.

To account for these factors, we designed three new complexity metrics that quantify the lines of code that demonstrate the above complexity. These measure respectively the number of lines of code that (i) make references to kernel-defined structures and functions (*kernel reference*), (ii) get invoked in *interrupt context*, and

(iii) synchronize or belong within a *critical section*. For each complexity metric we report the absolute values and the percentage reduction.

5.5.2 ABLE Implementation Details

We implemented ABLE as a kernel module for the Linux kernel version 2.6.24. We start by reporting in Figure 5.7 the lines of code and cyclomatic complexity metrics per component of ABLE itself. The ABLE code-base currently totals 2274 LOC.

Component	LOC	CC
I/O Router	92	16
ABLE Primitives		
I/O Library	528	74
Workload Profiler	195	35
Resource Manager	1074	96
Device Manager	166	21
Extension Registrar	34	2
Extension Profiler	65	8
Reasoning Engine	120	18

Figure 5.7: ABLE component statistics.

5.5.3 Simple RAID-1 Illustration

We implemented vanilla and ABLE versions of a reduced functionality RAID-1 system, which we call *Simple RAID-1*. Simple RAID-1 only provides write request mirroring (with retry for failed write operations) and read dispatching based on closest head position. Drive reconstruction is not addressed.

Figure 5.8 presents the functions implemented (with high-level pseudo-code) in the Simple RAID-1 system, along with a comparison of LOC used in each version (other complexity metrics are presented later). ABLE provides full support for most of the functionality required; in such cases only an ABLE API call is necessary. To implement mirroring write requests (as implemented by function *handle_write*), the

Functions (with pseudo-code)	ABLE support	Vanilla LOC	ABLE LOC
handle_write:			
Create new (cloned) I/O request	∼	34	2
Stack completion functions	✓	6	1
Issue I/O request	✓	1	1
write_complete:			
Check if unsuccessful	□	1	1
Reissue write	✓	124	1
Report write success	✓	2	0
handle_read:			
Obtain drive head position	✓	16	2
Obtain drive head direction	✓	16	2
Calculate closest head	□	13	13

Legend: ✓full support, ∼ partial support, □ no support

Figure 5.8: Function pseudo-code and LOC comparison for Simple RAID-1 extension.

vanilla version must create a full copy of the block I/O write request and assign the mirror device as its destination. The I/O completion handler (*write_complete*) must be stacked on top of the default I/O completion handler by explicitly modifying structures associated with each request. The new I/O request is then dispatched while the original request continues along its default path.

The *write_complete* function is invoked within the I/O completion interrupt handler for write requests. Checking if a write request failed requires a simple check of a variable; if yes, the I/O request must be reissued. However, this function runs in interrupt context and does not allow blocking operations. We addressed this issue by having a dedicated kernel thread that resends failed write requests, funneling request information to this thread via a shared FIFO queue. This increased the complexity of the vanilla version.

In the ABLE version, creating new I/O requests is supported by an ABLE primitive that clones all the fields of an I/O request; the only additional task for the developer is to manually modify the target device. Stacking a completion han-

dler and issuing the new I/O request also used ABLE provided primitives. ABLE provides an API to submit I/O requests from interrupt context, thus drastically simplifying the ABLE version.

For handling read requests (function *read_complete*), we implemented a greedy strategy to select the head closest to the request. The ABLE version uses the approximate head position and direction collected by the ABLE device manager. However, for the vanilla version this information is not available, hence a method for collecting device head position and direction was needed.

Even though the Simple RAID-1 functionality is very easy to describe and understand, the actual vanilla implementation used more than 450 LOC and yielded a high complexity (discussed next). Further, most of this complexity comes from pieces of code that handle complicated kernel level operations which are easily encapsulated within ABLE's library functions.

5.5.4 Quantitative Complexity Evaluation

We implemented four extensions to more broadly evaluate the ABLE infrastructure. The RAW extension performs a Read-After-Write to verify writes to medium. The Encryptor extension simply uses the built-in kernel DES implementation to encrypt written and decrypt read data. The Borg extension implements a block reorganizer on disk based on observed access patterns [BGU+09]. The Exces extension performs caching of popular disk data in a low-power flash device to power-down the disk and save energy [UGB+08]. The Borg and Exces extensions both include substantial non-kernel components for performing data mining; thus, we distinguish between the full version and a version that only accounts for code dependent on core kernel functionality.

Extension	Lines of Code			McCabe's CC			Kernel Reference			Interrupt Context			Critical Section		
	Vanilla	*ABLE*	*Reduction*	*Vanilla*	*ABLE*	*Reduction*	*Vanilla*	*ABLE*	*Reduction*	*Vanilla*	*ABLE*	*Reduction*	*Vanilla*	*ABLE*	*Reduction*
Simple RAID-1	473	115	76%	109	12	89%	95	10	90%	145	54	63%	88	12	86%
RAW	184	97	47%	19	7	63%	44	20	55%	71	7	90%	16	0	100%
ENCRYPTOR	121	107	12%	10	8	20%	15	10	33%	13	13	0%	4	0	100%
BORG	2888	2564	11%	631	567	10%	271	147	46%	127	51	60%	129	71	45%
BORG-KERNEL	1255	913	27%	201	137	32%	271	147	46%	127	51	60%	129	71	45%
EXCES	1114	946	15%	186	145	22%	230	126	45%	69	43	38%	175	125	29%
EXCES-KERNEL	761	593	22%	118	77	35%	230	126	45%	69	43	38%	175	125	29%

Table 5.3: Development complexity statistics for extensions.

Table 5.3 presents the five complexity metrics introduced earlier for all the extensions. The LOC required to develop the extension using ABLE were fewer by 11% (for BORG full-version) to 76% (for Simple RAID-1). Reduction in CC was noticeably higher than reduction in LOC for all extensions, except BORG vanilla, indicating that in most cases cyclomatically complex code was eliminated in favor of non-complex code. More importantly, when we examine the three metrics addressing kernel development complexity metrics, even greater reductions are apparent. Reduction in LOC involving kernel references ranged from 33% (for Encryptor which uses a substantial kernel DES encryption algorithm for both variant implementations) to 90% (for Simple RAID-1). Reduction in LOC involving interrupt context ranged from 0% (for Encryptor) to 90% (for RAW which does substantial processing in the write and read I/O completion handlers). Critical section LOC was reduced by 29% (for the EXCES variants) and were completely eliminated for the RAW and Encryptor extensions.

These findings indicate that the ABLE infrastructure can help developers by simplifying their development effort for a variety of self-management extensions.

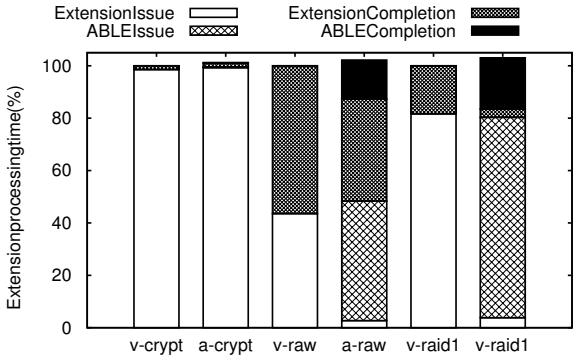


Figure 5.9: Normalized extension processing times. Prefixes V and A refer to the vanilla and ABLE versions respectively. Results are averaged over five runs.

5.6 ABLE Runtime Overheads

We quantify the runtime overhead that ABLE-based extensions incur by examining two extensions – ENCRYPTOR representing the lowest use of the ABLE API, and two RAW and Simple RAID-1 which make extensive use of it. We benchmarked overhead using PostMark configured to do 1000 transactions on files of size varying from 1 MB to 10 MB, on an Intel Pentium 4 2.00 GHz with 512MB of RAM and three 20 GB MAXTOR 6L020L1 hard drives running Linux 2.6.24. Figure 5.9 depict *normalized extension processing time*, accounting only for the times spent within the extension logic (including within ABLE API invocations) and excluding time spent in the rest of the I/O stack such as application logic, file system logic, and device I/O time. Notice that when the ABLE API is barely used (ENCRYPTOR), most of the time is consumed by the extension and ABLE only adds a 1.25% overhead. However, even when the extension logic mostly involves ABLE's API invocations (upto 83% of extension processing time spent within ABLE library functions for Simple RAID-1), there is only a 2.18% and 4.46% increase in total extension processing time for RAW and Simple RAID-1 respectively. Further if we compare the extension runtime with the I/O time for the benchmark, ABLE only adds a 0.38%, 0.08% and 0.14%

respectively for the ENCRYPTOR, RAW and Simple RAID-1 extensions. These numbers indicate acceptable overheads attributable to functionality encapsulation with ABLE.

5.7 Summary

The ABLE project develops an evolvable block layer infrastructure that substantially reduces developer effort while implementing self-managing storage systems. This infrastructure achieved a reduction of 11% to 76% in LOC and 10% to 89% in CC metrics across five block-layer extensions that we implemented, while incurring in less than 1% runtime overhead. Average reduction across the three new metrics designed specifically for measuring kernel complexity were even higher, averaging a reduction in the range of 22% to 93%.

ABLE also develops a novel theory of block layer storage extensions that creates a logic framework within which the extension stacks can be analyzed; this substantially simplifies the task of systems administrators who compose storage systems using self-management extensions as building blocks. This stacking theory is meant only to serve administrators decide on stacking extension orders, choosing which extension to deploy is out of reach of this theory since it requires to reason about extension effectiveness. In the following chapter we address an instance of this problem, in which there are two extensions with the same high level goal and the administrator must chose which one to deploy using an experimental based approach.

CHAPTER 6

COMPARING EXTENSIONS

In Chapter 5 we discussed a formal mechanism to determine appropriate extension stacking orders to meet an administrator defined goal. We presented use cases where ABLE helped administrators pick a stacking order for a set of storage extensions to accomplish a desired goal. However, this formal reasoning is insufficient in choosing among extensions that accomplish the same goal. Such decision is dependent on both the internal behavior of the extension as well as the properties of the workload. Capturing the complex characteristics of internal extension behavior and the workload all within a formal reasoning process seems exceedingly difficult.

In this chapter we present an empirical methodology to compare and contrast two extensions that storage administrators can use to achieve the same goal. This approach takes into account both workload and extension characteristics in the comparison. In particular, we focus on comparing two extensions that reorder data between storage tiers one using a *caching* based approach and the second *multi-tiering* (also known as tiering), we explain both techniques bellow. To determine which extension is more appropriate for a given workload we will first study the workload (6.2), next describe the inter workings and variables that affect the performance of both systems in 6.3; later, in 6.4, we evaluate both systems under a set of axis to determine with is most appropriate under a given load.

6.1 Problem Background

There is substantial interest in both the industry and academia about ways to integrate flash-based storage into existing disk-based storage systems due to their complementary cost, performance, and power characteristics. There are two primary camps or schools of thought about doing flash storage integration:

- **Caching** argues for managing flash-based storage as a large caching layer in the storage hierarchy to capture the working set of the data stored in the disk layer.
- **Multi-tiering** argues for managing SSDs as the front-tier of the disk drive based storage, as a primary data store.

Both camps are well represented in industry solutions. Some of the solutions with caching include NetApp's FlashCache [Pet09b], Oracle (via Sun) ZFS storage appliances 7000 series [Ora10], and Nimble storage CS-series [Nim10]. Similarly, some of the storage vendors with multi-tiering based storage systems include IBM [Tan10], EMC FAST [Lal09], 3PAR [Pet10], Compellent [Pet09a], and Avere [Ave10]. Both camps claim to provide performance gains and cost savings by using SSDs as part of the storage layer.

The plethora of solutions in both of these classes (caching and tiering) has led to online debates about the superiority of each technique for enterprise workloads [Mar10, Owe10]. Even within the same technique, multiple variants exist. For instance, while NetApp FlashCache favors using flash-based storage as a read-only cache, Oracle's Unfied Storage solution uses SSDs to cache both reads (L2ARC) and writes (ZIL). Tiering solutions from different vendors also operate at different time and space granularities — hourly vs. daily migration and whole volumes, files, or large chunks migrations — leading to very different performance outcomes for the same workload.

Meanwhile, many enterprises are yet unsure of how to make best use of SSDs in their environment. Comprehensively characterizing each technique and understanding the impact of various parameters on storage performance is critical for efficient use. For example, in case of caching, the cache replacement technique needs to be aware of the specific characteristics of SSDs in terms of reads vs. writes as well as

new load imbalance issues (see Table 6.1). This is quite different from previous main memory caching solutions where it was sufficient to exclusively focus on improving the hit rate.

Level	Access Time		
	Random Read	Random Write	Sequential Read/Write
Register	< 1 ns	< 1 ns	< 1 ns
Level 1 Cache	2 ns	2 ns	2 ns
Level 2 Cache	7 ns	7 ns	7 ns
Level 3 Cache	15 ns	15 ns	15 ns
DRAM	45 ns	45 ns	45 ns
OCZ PCI-e SSD	25,000 ns	1,500,000 ns	1,000,000 ns
Intel X25-M SSD	75,000 ns	4,000,000 ns	2,500,000 ns
SATA Disk	7,000,000 ns	7,000,000 ns	2,000,000 ns

Table 6.1: Access times throughout the memory hierarchy. Numbers reported are approximations taken from the device specification sheets.

We start off with a basic description of caching and multi-tiering based solutions. This is based on the existing literature as well as our own experience in building a multi-tiering solution in the recent past [GPG+11]. We then compare these two approaches across various performance dimensions such as their adaptability to workload changes, SSD effectiveness, leveraging of device heterogeneity, metadata overhead, and reliability. For this comparison, we have used real workload traces and both a simulation and Linux based implementation of each approach. Our goal in this chapter is to present an empirical methodology that gives insights into how would an administrator choose tiering over caching, or vice-versa. More concretely, we seek to answer the following two questions:

1. *Are caching and tiering fundamentally different in how they operate and what they can achieve or are they, in fact, two sides of the same coin?* In other words, by controlling the tunable parameters in each solution class carefully, can we make caching approaches lead to tiering-like outcomes and vice-versa?

2. *If caching and tiering are fundamentally different in their operation, which solution best under which circumstances?* In other words, how do we tell when is better to deploy one solution over the other? And equally as important, how should we configure such solution to achieve the best performance for the given load?

6.2 Understanding Workload Characteristics

Before we decide on design aspects of the caching and multi-tiering solutions we analyze two real life storage system workloads. We study traces collected by researchers at Microsoft Research Cambridge [NDR08], it contains block level I/O traces of 36 independent NTFS volumes of diverse types used within an industrial research lab. The second trace we study was obtained from a storage server at the FIU computer science department, this storage server contains user's homes and email for faculty and graduate students. Next we present some of the workload characteristics we found in the traces.

Trace	**Length**	**# I/Os(x10^6)**	**Vol. Size**	**% Accessed**	**R/W Ratio**
MSR	7 days	433	6.12 TB	53%	3.21
FIU	7 days	697	20 TB	48%	3.99

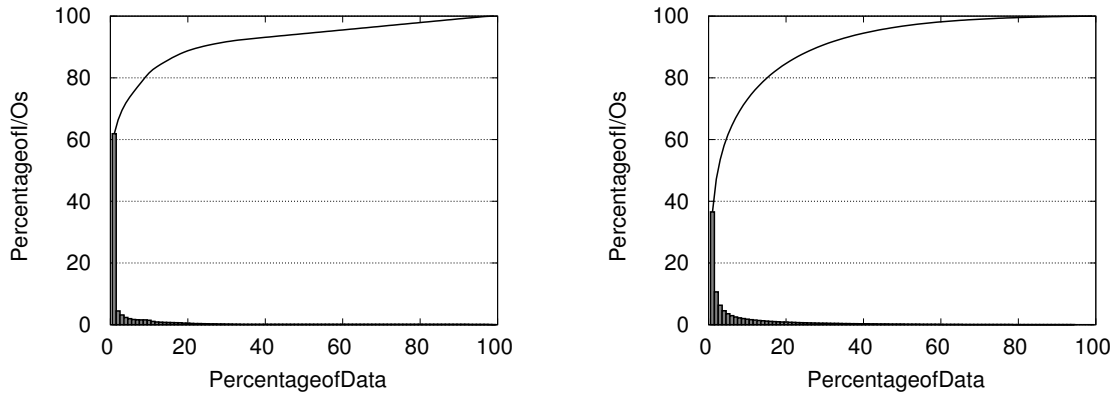
Table 6.2: Summary of Traces Studied

Table 6.2 summarizes some of the characteristics of the traces. In both cases around half of the total space is accessed in the period observed. This amount might be considered high compared to data reported in previous studies [BGU+09, GS02]. We attribute this to two factors. First, since we don't have information regarding the original size of the volumes thus we are relying on the last block accessed to calculate their size which may possibly be an underestimate. Second, it might be possible that these traces contain access made by maintenance tasks, such as backups and

RAID scrubbing. These tasks typically access all data in the volume, leading to an inflation on the percentage of data accessed. The other important factor is that both workloads exhibit a read intensive pattern, more than 75% and 80% of the I/Os are reads for the MSR and FIU workloads respectively. This is a point in favor of adding SSDs to the storage since they have significant lower read latencies than HDD, as seen in Table 6.1.

Systems that incorporate different types of storage are designed with two assumptions in mind. The first is that I/Os are not distributed uniformly across the data. Previous studies [BGU+09, RW93a, GS02, HSY05] have pointed out that not all portions of data receive the same I/O load. It is common to find workloads that exhibit a logarithmic distribution of the I/O load, with a small subset of the data receiving a great number of I/Os and the remaining I/Os spread through a large amount of data. If this assumption holds true for a given workload, then systems that leverage multiple storage tiers (such as SSD and SATA disks) are very likely to be more cost effective than single tiered systems for the workload.

The second assumption is that I/O load is not constant but varies though time. This assumption is based on the observation that many systems are not utilized uniformly across time. Two well known examples that produce periods of high I/O activity are bootstorms (when many computers boot simultaneously at the start of the work day), and batch processes (e.g. backups) where large amounts of data are processed sequentially. However these periods typically do not span more than a couple of hours. Hence, systems that leverage multiple storage tiers offer a great potential at improving performance while requiring less physical resources. Next, we examine the workloads to see if these characteristics are present.



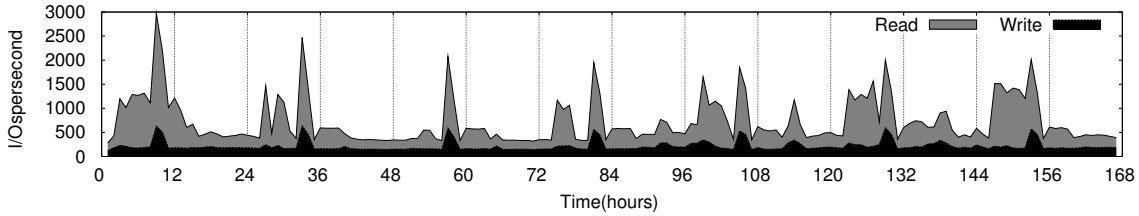
(a) MSR (b) FIU

Figure 6.1: I/O distribution through the data. The bars depict the % of I/Os the issued to the top X% of data accessed. The solid curve depicts the same information as a cumulative distribution.

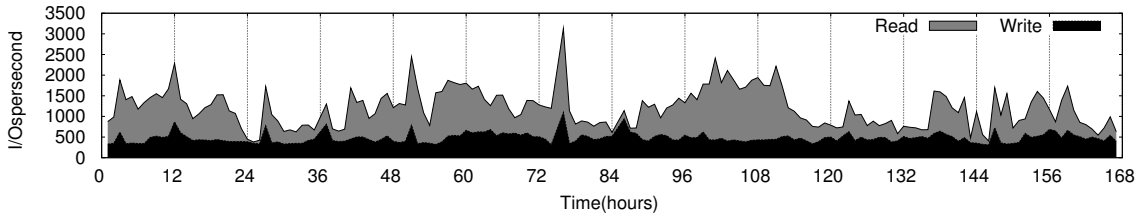
6.2.1 Storage I/O Distribution

It is well know that stored data is accessed non-uniformly [RW93b]. In Figure 6.1(a) we see that for the MSR workload the top 1% (which equates to 3 GB) of the data accessed gets 62% of the entire week's I/Os load. Additionally we notice that the top 20% of data contribute towards more than 90% of the accesses. For the FIU workload we observe similar trends. In this case more than 35% and 85% of all I/Os are to the top 1% and 20% of the data respectively. This shows a very skewed pattern in which a very large portion of the load could be serviced using a small fraction of the space.

This data supports our assumption that I/Os are not distributed uniformly across the data and motivates the need for dynamic heterogeneous systems, which have been shown to reduce cost and energy as shown in chapter 4.4.



(a) Microsoft Research Trace (MSR)



(b) FIU Computer Science Dept Trace

Figure 6.2: I/O operations per second across time for the MSR and FIU workloads.

6.2.2 I/O Distribution Through Time

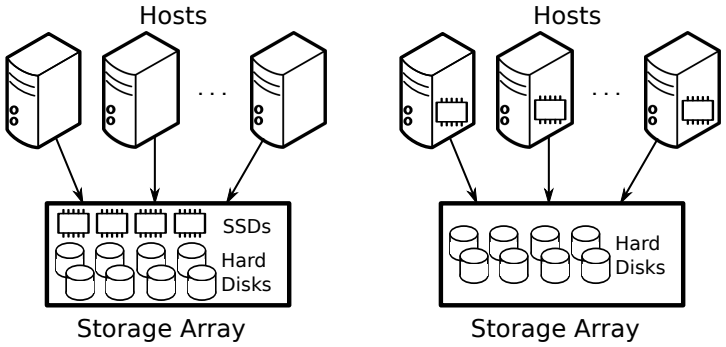
Workloads that have variability in I/O activity and skewed data access distribution are good candidates for multi-tiered storage systems. These systems can benefit from the periods of low activity in multiple ways. For instance, they can use this time to destage data from the SSD tier into the HDD tier, so that the SSD has a reserve of free space to mitigate peaks in load.

Figure 6.2 depicts the I/O load through time for both workloads. For the MSR workload we observe a couple of interesting behaviors. First, there are 12 hour periods of "high" load (e.g. 0-12, 24-36), and followed by other 12 hour periods of "low" load (e.g. 12-24, 36-48). Further study in [GPG+11] reveals that high and low load periods correspond periods of predominantly sequential and random I/O activity respectively. Second, we observe that the highest peak period (evidenced in hours: 6, 30, 54, etc) repeats every 24 hours. For the FIU workload is harder to detect any visible pattern. However, we do see that the workload has a mix of long periods of high I/O activity and short periods of low activity. These observations

support our second assumption that I/O load is not constant, and varies through time.

6.3 Overview of Two Approaches

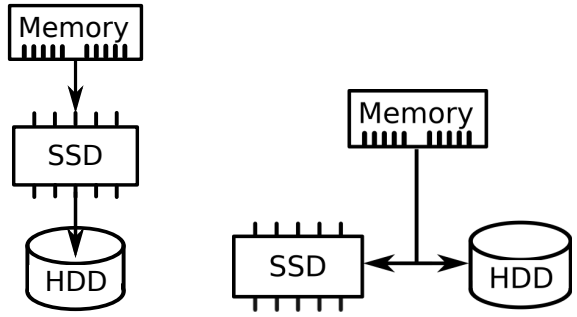
In this section, we present an overview of SSD integration solutions which follow either the caching or the multi-tiering approach. These solutions are based on the well known literature in both areas and serve to illustrate both the strengths of each approach as well as the baselines that we evaluate in later sections. To put these solutions in context, it is important to understand the usual assumptions behind flash integration solutions. Figure 6.3 illustrates two system models for flash (SSD) storage integration. In Figure 6.3(a), SSDs are part of a networked storage array and thus shared across hosts, whereas in Figure 6.3(b), each host (storage client) has a local SSD. In both cases, SSDs can be either used for caching or tiering. We refer to these topologies as *shared SSDs* and *local SSDs* respectively. In this study we focus on evaluating the design space for shared SSDs solutions. Next, we discuss the design choices for our implementation of both caching and tiering solutions.



(a) Shared SSDs (b) Local SSDs

Figure 6.3: System architecture with (a) shared and (b) local SSDs

Figure 6.4 depicts the high level architectural diagrams of both caching and tiering. In abstract terms, caching populates the SSD while the I/O is in-flight, that is, when the I/O operation is underway. Tiering on the other side, collects statistics during a given time frame and later migrates data to the SSD based on the observed I/O access pattern. For the results presented in this and later sections we implemented both a caching and tiering systems in Linux, additionally we also implemented a caching and tiering simulator to allow us to conduct long term analysis on system behavior. For all experiments in this section both systems were configured using 2x120 GB Intel 320 Series SSD configured as a RAID 0, and 12x1 TB ST31000524NS SATA disks also configured as a RAID 0.



(a) Caching (b) Multi-tiering

Figure 6.4: Architecture diagrams of the proposed caching and tiering schemes incorporating SSDs.

6.3.1 Caching

Caching is based on maintaining a portion of the workload from a slow (and typically much larger) memory device within a faster and smaller memory device, in anticipation that such data will be used again in the future and those later accesses be made faster. Multiple variants of caching are already incorporated in different layers of the system, for instance between CPU and DRAM.

In caching systems every time an I/O is received we check the upper level memory or storage layer (also known as the cache) to see if the data is present. If present, data is serviced from it and a *hit* is reported. If false, also called a *miss*, an I/O is issued to the lower storage layer to fetch the data. If the cache is full, we would need to evict old data from it before we are able to bring in the new data. Selecting the data to be removed is the task of the caching algorithm. Typically every entry in the cache (chunk) has the same size.

There are many factors that affect the performance of caching systems, in this work we focus on analyzing four, namely: cache size, chunk size, caching algorithm, and write policy.

Caching Algorithm

There is a very large number of known cache replacement algorithms (or caching algorithms). Researchers have proposed using frequency based replacement algorithms for second level caches arguing that recency is mostly captured by the first level cache leading to the inter-reference gaps being much higher for second level caches, such algorithms include LRU-K [OOWZ93], 2Q [JS94], LIRS [JZ02], ARC [MM03], and MQ [ZCL04]. Using these type of algorithms the overall space consumed by metadata maintained by a cache replacement algorithm is typically of the order of *cache size + the number of ghost buffers* (used to track chunks that are currently not in the cache but were accessed in recent past).

We implemented and tested LRU which represents the simplest and most widely used caching algorithm, and MQ which represents an algorithm specifically designed for storage level caches. MQ classifies the chunks into different queues depending on the amount of times it is accessed. Chunks can be promoted to a higher queue if its access counter increases and demoted to a lower queue if it is not reuse promptly. We

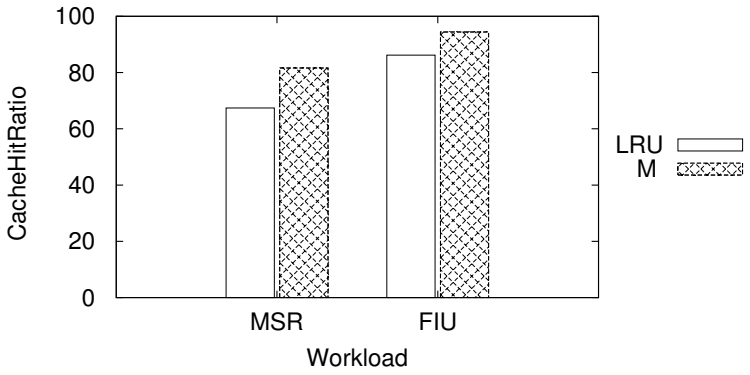


Figure 6.5: Hit rate of LRU and MQ caching algorithms for the 7 day period, using 16 KB chunks and a 100 GB cache.

use the same algorithmic parameters as those presented by the authors in the MQ paper[ZCL04], namely 8 queues plus a ghost buffer of 2X the size of the cache and $log(\#\ \text{access})$ to assign items to queues on access. Figure 6.5 depicts the average hit rate and response time of LRU and MQ for different periods of the MSR workload. In all cases the MQ algorithm was able to achieve a higher overall hit ratio.

Cache Size

One of the simplest way to improve the effectiveness of a cache is by increasing its size. Bigger caches are able to capture a larger portion of the workload, and thus incur more hits thus a higher hit ratio. However, as we saw in Figure 6.1 the distribution of I/Os to data is very skewed, with a large percentage of the I/Os focused on a very small portion of the workload. This property implies that with a relatively small cache we might be able to service a significant portion of the workload. But, after some point, we would need increase the size of the cache significantly to obtain a noticeable increase in performance.

In order to the quantify the relation between cache size and hit ratio we ran both workloads through a cache simulator based on the LRU algorithm. Figure 6.6 depicts the hit ratio for both the MSR and FIU workloads when using 16 KB chunks

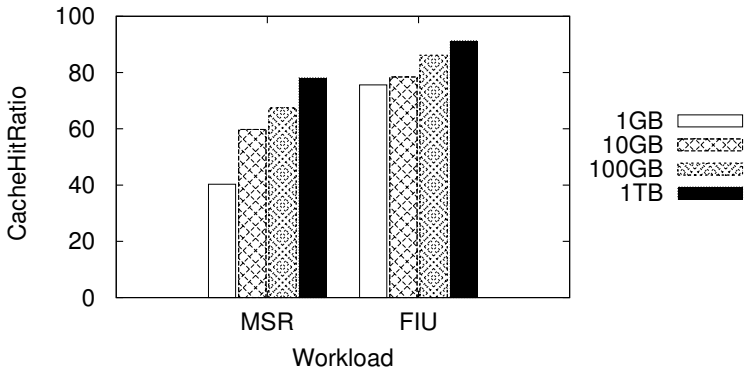


Figure 6.6: Hit ratio for different cache sizes.

and varying the cache size. For the MSR workload there is are noticeable jumps in hit ratio with every as we increase the cache size, the highest increase come from increase of 1 GB to 10 GB and 100 GB to 1 TB. For the FIU workload we see no significant difference in the hit ratio as we increase the cache from 1 GB to 100 GB, only after 100 GB the hit ratio starts to increase in a perceivable way. However, it is important to point that in this experiment we increased the size of the cache by 10X each time. Such increase will likely come at a significant capital cost.

Chunk Size

Another important factor for dynamic heterogeneous storage systems is the size of the unit of data at which it operates. For caching systems this unit of data is referred to as chunk size. The size of the elements of the cache will have a direct impact on its effectiveness. Intuitively small chunks can potentially be more efficient by only storing precisely the data that gets accessed, but they will incur higher metadata overheads which could potentially complicate its reliability. On the other hand, bigger chunks will be have less metadata overheads but might provide reduced hit ratio due to a less effective utilization of the cache. Therefore, we should pick a chunk size that balances both adaptability and metadata overhead.

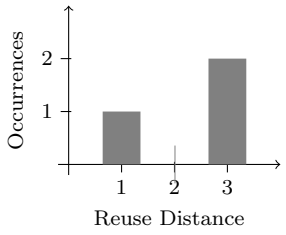
The best way to see the impact of the chunk size would be to run multiple instances of the algorithm varying both the chunk size and cache capacity. Previously, researchers have simulated LRU with an unlimited cache to obtain the hit ratio curves [MGST70, ZPS+04], but this doesn't scale well to storage sized caches as it is slow and requires a significant amount of memory.

One alternative approach to implement LRU is to use count for every access the number of *unique* chunks referenced between now and the last time it was access (including itself). This is typically know as *reuse distance*. For instance in Figure 6.7(a) the reuse distance between the first and second accesses of A is 2 and between the second and third access of B is 3. The important observation here is that a distance of X indicates that given a cache of at least X items one would get a hit for items with reuse distance of X or lower. But again, calculating the number of *unique* chunks accessed requires us to maintain a stack of all previous accesses and traverse the stack for every I/O searching for the previous occurrence of the item counting the number of distinct items accessed.

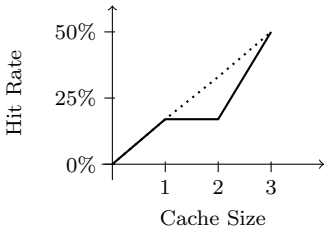
Cache Size	Accesses					
	A	B	B	A	C	B
1-item	×	×	✓	×	×	×
2-item	×	×	✓	✓	×	×
3-item	×	×	✓	✓	×	✓

Legend: ×miss ✓hit

(a) Access Sequence



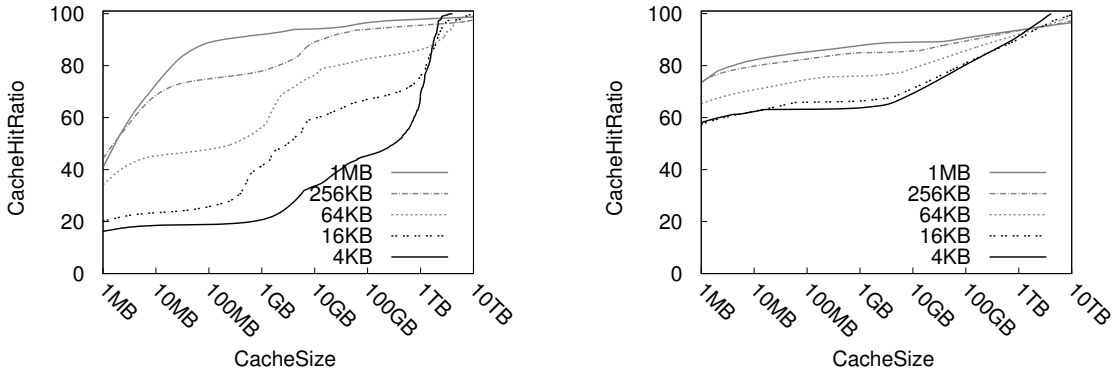
(b) Reuse Distance



(c) Hit Ratio Curve

Figure 6.7: Obtaining the miss hit rate curve from an access sequence. (a) Access sequence and actual hit ratio for caches of different sizes, (b) Reuse distance of the sequence, (c) Hit ratio estimate calculated using the reuse distance, dotted line indicates actual hit ratio.

If we were to relax the definition of reuse distance and only count the number of accesses instead of the number of unique chunks accessed, the computation and memory requirements would be significantly less since we would only need to maintain information about when particular chunks where last accessed. Now going back to the example in Figure 6.7(a), using our relaxed reuse distance metric we get that the distance between the first and second accesses of A is 3. It's easy to see that this would likely introduce error in the hit ratio calculation, leading to an overestimation of the amount of cache required (as seen in Figure 6.7(c)). However, overestimating the size of the cache will not result in any performance degradation and may in fact improve performance by augmenting the hit ratio. The negative part of overestimating the cache size is the increased capital expenditure to acquire the extra cache storage. If this is a concern we can simply use this technique as a reference to get an initial estimation and later use simulation to obtain the actual hit ratio for a given size.



(a) MSR (b) FIU

Figure 6.8: Hit ratios for different cache line sizes for the MSR and FIU workloads. Lines depict the cache line size.

Having found an efficient way of calculating the hit ratio curves we ran both workloads through the cache simulator using different chunk sizes to obtain the hit ratio curves, which are depicted in Figure 6.8. Notice that for cache sizes smaller

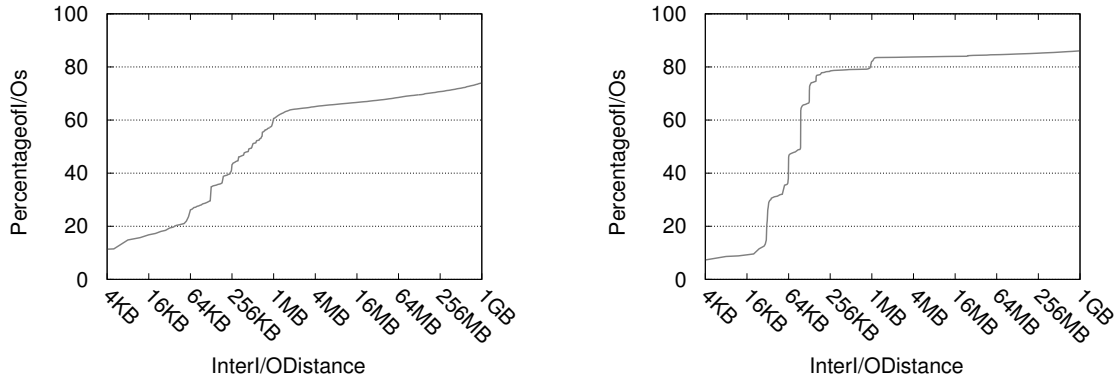
than 1 TB having 1 MB chunks will provide the highest hit rate at a very low metadata overhead. This finding can be corroborated in Figure 6.9(a), where we see that $> 60\%$ of the I/Os fall within a 1 MB region of previous one, thus using larger chunks benefits from the locality of access in the workload. For caches larger than 1 TB, 4 KB chunks provided the highest hit rate. However, at a 100X increase in metadata overhead compared with 1 MB chunks.

These findings contradict our intuition that smaller chunks will always provide a higher hit ratio. However, there are two complementary reasons for this. First, the working set is large, thus there is a very low probability that the same data will be requested in a short time spawn. Second, there is a significant amount of spatial locality. Hence, while the chance of accessing the exactly same the 4 KB piece of data in the near future is small there is a high probability to access data that is close (in terms of LBA distance).

Figure 6.9 shows the cumulative distribution probability of the distance between two I/Os (in terms of LBA). We observe there is a small ($< 10\%$) probability that the next I/O is to exactly the next 4KB page, but if we consider a range of 1MB we see the probability of a hit jump to 60% and 80% for the MSR and FIU workloads respectively. This suggests that a majority of the I/Os will fall within 1 MB the previous I/O. Thus, we observed higher hit ratio when using 1 MB chunks because we were, in a way, prefetching data that is going to be accessed in the near future.

Write Policy

Given the asymmetric I/O and write endurance properties of the SSDs, many researchers have expressed concern about the potentially shorten lifespawn of SSDs when used as caches for workloads that experience a write intensive load. SSD manufacturers have responded by providing strong minimum lifespawn guarantees for



(a) MSR (b) FIU

Figure 6.9: Distribution of inter I/O distance for the MSR and FIU workloads.

their products, in many cases recurring to throttling writes when the workload is write heavy [LKKK12]. Some researchers have even proposed extending the lifetime of SSDs by using disks as write caches [SPBW10].

In this work we consider two basic write policies for the SSD cache. When using the *write back* policy both reads and writes are treated equal. This approach might hurt performance for write intensive workloads and shorten the lifespawn of SSDs for write intensive workloads. On the other hand with the *write through* policy, all write I/O is diverged to the disk, after the write completes the I/O is reported as done and a new write sent to the SSD. However, using the write through policy can also hurt performance since an extra I/O is generated for every write.

To measure the performance difference between both write policies we replayed two different periods of the MSR trace. For the hours 6 to 12, that represents the period of highest I/O activity and also that of a high read to write ratio, write through has 9% lower average service time. In this case the performance advantage comes mainly from the fact that eviction in write through case are "free" since all data in the cache is always clean. But, for a period where the read to write ratio is low, represented in hours 42 to 48 of the trace we notice how the write amplification

of the write through policy negatively impacts performance resulting in 15% worst performance compared to write back.

6.3.2 Multi-Tiering

Multi-tiering involves periodic data migrations among the available storage tiers to accommodate for changes in the workload. Upper tiers correspond to faster but smaller tiers, while lower tiers are bigger but slower. Typically, data is initially placed in a lower tier and promoted or demoted based on its access pattern. In such cases data can either be replicated or moved across tiers. In this study we focus in the case when data is moved, meaning there is a single copy of the data active at all times. The granularity of movement is a chunk of data (sometimes referred to as an extent) of the order of megabytes in size. We do not consider volume-based multi-tiering in this study which is relatively unattractive for primary storage [GAW09].

Contrary to caching systems, in multi-tiering chunks are not migrated on every I/O, rather a set of statistics are collected (per chunk) and later analyzed. Such analysis yields a possibly new chunk to tier assignment, which indicates the chunks that need to be migrated between tiers. We refer to the time lapse between successive analysis/data-migration operations as an epoch. The length of epochs range from a few hours (30 minutes in the EDT system [GPG+11]) to an entire day across commercial solutions.

As with caching, many factors affect the performance of multi-tiering systems this work focuses on studying: the multi-tiering algorithm, chunk size, and reconfiguration interval.

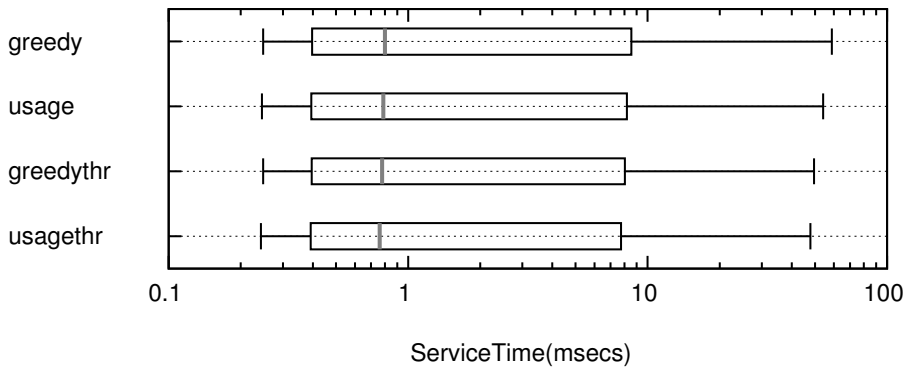


Figure 6.10: I/O Service times for the tiering system varying the algorithm during hours 6-8 of the MSR workload, using 16MB chunk and 30min epochs.

Multi-Tiering Algorithm

Tiering relies on figuring out a chunk placement that improves the current performance. The tiering algorithm decides chunk placement within individual tiers, based on chunk level statistics collected in an online manner over several epochs. For this work we considered two approaches. First, placement based solely in IOPS, which we term *greedy*, where chunks are sorted by IOPS and assigned to the highest (fastest) tier with free space. Our second approach is taken from EDT [GPG+11] and is based in per chunk resource utilization. The idea is to place each chunk in the tier where it utilizes the least amount of resources. Here utilization is a multi axis metric that includes space, IOPS, and throughput. We name this approach *utilization*. Figure 6.10 depicts the 5^{th}, 25^{th}, 50^{th}, and 95^{th} percentiles of I/O service times. Notice that both algorithms perform very similarl. However, the greedy algorithm incurs in longer latency for the slower I/O (as seen by the 95^{th} percentile). This leads the utilization algorithm to perform 10% better on average in this case.

However, given that tiering only changes the chunk layout after epochs, we added a mechanism that identifies chunks that are over demanding resources from their present tier and migrates them to a different tier that can account for its I/O de-

mands. We refer to this mechanism as throttling detection. We see in Figure 6.10 that the algorithms with throttling detection enabled (identified with the "+thr" suffix) reduced the maximum latency in both cases. In particular, for the utilization algorithm adding throttling detection improved average response time by 12%. Therefore, for subsequent tiering experiments we will use the utilization based algorithm in combination with throttling detection.

Chunk Size

Multi-Tiering solutions have argued for a fixed chunk granularity within each solution [Sil11]. Since a chunk metadata access is involved in the data path, it must be cached in memory to minimize the overhead and its in-memory footprint must be controlled. Notice that we need to maintain metadata for each chunk of data, this number is in the order of $\sum_i$ *size of tier i*. Given a tiered storage system with a fixed amount of available memory to store metadata, the solutions in the literature suggest picking a large chunk size so that per-chunk metadata for all chunks would fit in the system memory [Tan10, Lal09]. However, larger chunks take longer to migrate. This makes the system slower to adapt to changes in the workload. On the other hand, choosing smaller chunks makes the system more adaptable, but increases metadata.

In order to evaluate the impact of different chunk sizes on the performance of the tiering system we trained the system with the first 5 days of the MSR workload to obtain a chunk mapping and replay the following 2 hours. Figure 6.11 depicts the response time distribution for the replay using various chunk sizes. We see that performance always improves as chunk sizes decrease with 1 MB chunks observing the best overall performance. This is mainly due to smaller chunks being able to

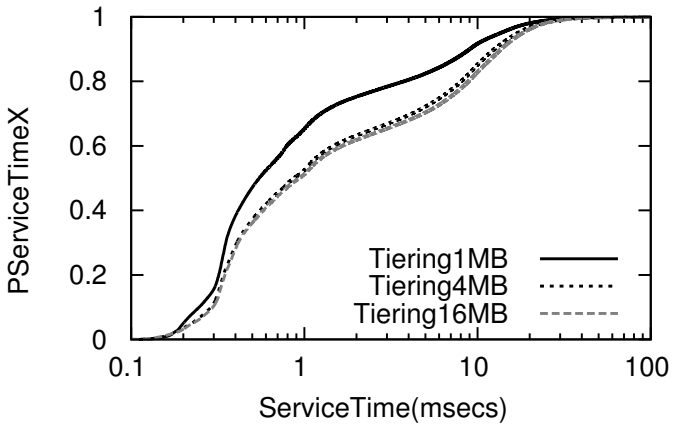


Figure 6.11: Cumulative distribution of response time for the multi-tier system during hours 120-122 of the MSR workload.

adapt quicker to changes in the workload by using less time to migrate between tiers. Similar behavior was observed when replaying other periods of the workload.

Chunk Size	Amount of Metadata (per TB)
64 MB	0.3 MB
16 MB	1.25 MB
1 MB	20 MB

Table 6.3: Amount of the metadata for the multi-tier. Assuming 20 byte per extent metadata.

As mentioned earlier we must also consider the metadata load imposed on the system. In Table 6.3, we see that the amount of metadata grows quickly. For instance, if we were to deploy our example multi-tier on system hosting 1 PB of data and use 1 MB chunks, we would use 20 GB of metadata. Although it is common to find servers with sufficient memory to host such amount of metadata, keeping it persistent and consistent may become complicated if the amount of metadata becomes too large. Additionally, allowing large amounts of memory to be used for chunk metadata will have negative performance side effects since less amount of memory will be available for other task (e.g. read caching or write buffering).

Epoch Length

The next parameter we study is the time between epochs. If this period is too long, then we might miss opportunities to improve performance. If is too short, we might incur an excessive number of migrations which could negatively affect performance. However, tuning the epoch length to the optimal value can be a difficult task. For instance the workload could be using different working sets at different times of the day. Further the length of these periods could also vary.

Many tiered systems opt to set the epoch length based based on the selected chunk size. The larger the chunk the more time taken to move it among tiers and thus systems using chunks in the order of hundreds of megabytes to gigabytes tend to have epochs in the 12 to 24 hour range. On the other hand, smaller chunks (tens of megabytes) will use shorter epochs ranging from tens of minutes to a couple of hours.

Figure 6.12 depicts the the 5^{th}, 25^{th}, 50^{th} and 95^{th} percentiles of response time for the tiering system when using 16 MB chunks. We see that larger epochs lead to better median response times. This is mainly due to having less percentage of time invested in data migrations. However, having less migrations makes the system more susceptible to workload changes. Thus, when we examine the overall mean response time we find the best performing epoch length to be 30 minutes.

6.4 Caching vs. Multi-tiering

In chapter 5 we presented a formal theory for storage administrators to compose extension stacks that achieve the desired goal. However, this theory does not allow the administrator to decide which extensions to deploy, it only proceeds once the extensions have been chosen. In particular, if the administrator disposes of more

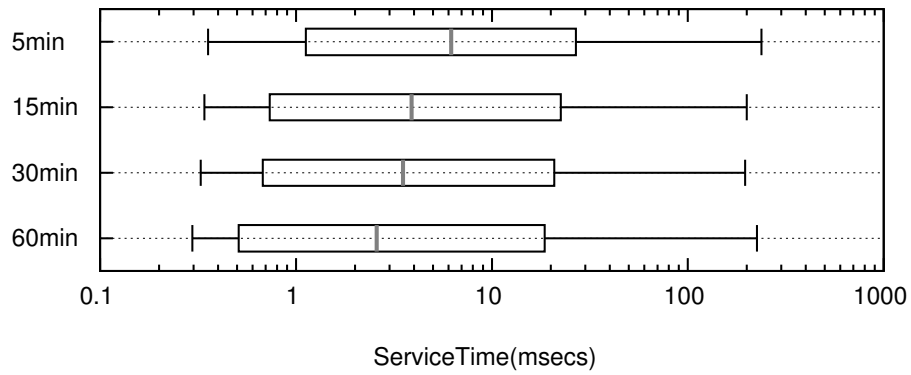


Figure 6.12: I/O Service times for the 5^{th}, 25^{th}, 50^{th} and 95^{th} percentiles for the tiering system varying the algorithm during hours 6-10 of the MSR workload, using 16MB chunk and varying the epochs length.

than one extension that achieve the wanted high level goal which one does she choose. To make this decision we would need to be able to deterministically determine how each extension acts to deduct which is better suited for the given workload.

In this section, we present an empirical methodology that allows us to gather more information on extension behavior. The study is based on comparing tiering and caching extensions across several dimensions such as workload adaptability, SSD effectiveness, leveraging device heterogeneity, metadata overhead, and reliability. For all experiments in this section we use a 200 GB SSD tier and 12 TB SATA HDD tier. In the previous section we studied the influence of different parameters on both systems, this allowed us to find a parameter configuration that is most adequate for a workload. Caching is configured to use the MQ algorithm with a write back policy and 1 MB chunks. Tiering is configured to use the utilization algorithm, 1 MB chunks and 30 minute epochs.

6.4.1 SSD Effectiveness

To quantify and contrast the effectiveness of the SSD device in both solution classes we used the MSR workload. In particular we use the first 5 days of the trace

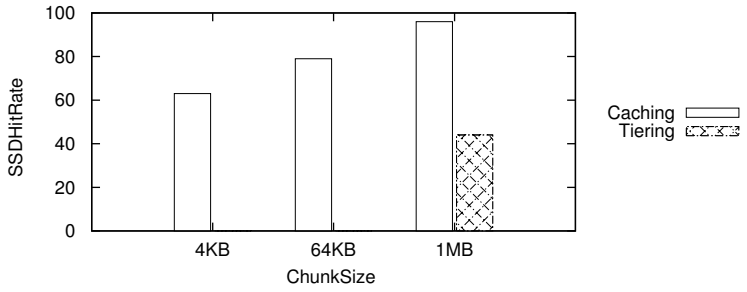


Figure 6.13: SSD hit ratio for a replay of hours 120 to 122 of the MSR traces using a 200 GB SSD.

to "train" the system (e.g. warm the caches, populate the tiers) and present the percentage of I/Os issued to the SSD for a replay of the hours 120 to 122 of the trace in Figure 6.13.

When interpreting the above result, is important to bear in mind that the goals of the caching and tiering solutions are quite different in principle. The primary goal with tiering is to meet performance objectives for data access choosing the best possible location for the data given long-term trends. With caching, the primary goal is to maximize the hit ratio in the higher performing SSD device. This is not necessarily optimal in terms of overall performance, in fact Wu, *et al.* [WR10] showed that better performance can be achieved by load balancing between the SSD and HDD tiers.

This is an important distinction from traditional DRAM caches which continue to improve performance at higher cache hit ratios because (a) the difference in performance between DRAM and the backing stores was always several orders of magnitude (irrespective of the nature of access, see Table 6.1), and (b) one DRAM access was mandatory anyway as data can not be read directly from disk to a CPU register. In case of an SSD cache, the performance difference between an SSD and a high performance (SAS) array backing store is not that significant for sequentially accessed data and in fact, using a $/MB/s metric, it is more optimal to have

frequently accessed sequential data in the SAS device instead [GPG+11, WR10]. Nevertheless, as SSD continue to get cheaper and improve performance we believe it will become the unconditional best choice for both random and sequential accesses. In fact, Table 6.4 shows that while HDD price to performance ratio has almost stagnated in the past two years, newer SSD models provide 2X the throughput and close to 10X more IOPS than previous models. The trends also suggests that hard drive will continue to provide the lowest cost per GB for years to come, thus leaving room for load-balancing approaches [WR10] to mitigate possible over-subscription issues.

Device	**2010**				**2012**			
	Cost	**$/GB**	**$/Mb/s**	**$/IOPS**	**Cost**	**$/GB**	**$/Mb/s**	**$/IOPS**
SSD	$430	$1.8	$1.46	$0.13	$180	$1.5	$0.78	$0.01
SAS	$325	$0.65	$1.62	$1.11	$310	$0.62	$1.55	$1.06
SATA	$170	$0.16	$1.61	$1.24	$110	$0.10	$1.05	$0.80

Table 6.4: Device Cost Comparison from 2010 to 2012. *SSD'10* is a 120GB Intel x25 MLC, *SSD'12* is a 120GB Intel 320 MLC, *SAS* is a 450GB 3.5in 15K RPM SAS, and *SATA* is a 1TB 3.5in 7.2K RPM SATA

6.4.2 Workload Adaptability

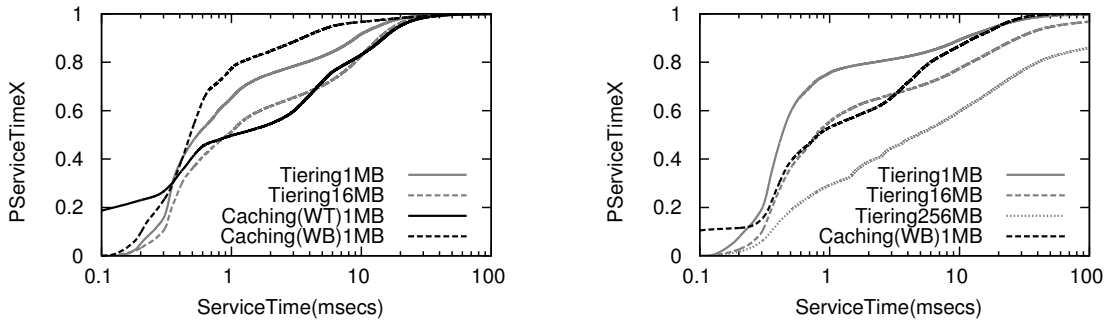
Cache replacement algorithms are designed to adapt quickly to a changing workload by bringing the current working set of data into the faster caching device on demand. However, data access patterns may involve longer term trends that are not easily captured using the limited (shorter term) memory. For instance, if typical *reuse distances* of hot data are large (e.g., cache lines are accessed hourly), a simple approach that only tracks the most recent block accesses may prove inadequate. In order to capture longer term trends, one can use *ghost buffers* which would increase the metadata requirement proportionately. Alternatively, one can use slow aging of cache lines. For example, MQ [ZCL04] algorithm uses a tunable *lifetime* parameter

to demote cache lines from higher frequency queues to the lower ones which allows for this optimization. Both of these techniques allow caching algorithms to potentially mimic the behavior of tiering solutions, that are discussed next.

Dynamic tiering algorithms are designed to perform data movement at the granularity of tens of minutes to hours to capture long-term trends in data usage. This design choice is driven by the chunk granularity of data movement, typically much larger than the line granularity employed in caching. Doing so implies substantial additional data moved for each data block accessed if conducted in-band. Consequently, out-of-band data movement is often used, in contrast to the in-band data movement in caching. To optimize data movement, tiering algorithms typically employ aging-based characterization of data access patterns to determine the most appropriate tier over a longer duration of time.

While long term stable trends are useful, tiering solutions can miss short term working set changes. Particularly, tiering solutions must add mechanisms to address short term hotspots and I/O bursts. For instance, the EDT system presented in 4.4 uses a throttling correction mechanism to alleviate unexpected hotspots on arrays that get temporally over-subscribed. However, such a reactive mechanism would incur a delay before throttling is detected and performance is restored via corrective actions. An alternate approach here is to provision for continuous data movement (as opposed to epoch-based) in tiering solution with quicker or dynamically chosen aging of extent characteristics. Further, as we shall argue shortly, chunk sizes can also be chosen to be smaller without substantially increasing the in-memory metadata footprint to perform data movement more efficiently.

Figure 6.14 shows the cumulative response time for two different workloads, we ran both caching and tiering algorithms using different chunk sizes. We we only include results for 1 MB for caching since they represent the best performance ob-



(a) Long term trends

(b) Change in working set

Figure 6.14: Distribution of response time for a (a) workload that exhibits a chance in its working set, and (b) one the honors long term trends.

tained. For tiering we show two chunk sizes which represent the overall trend. Figure 6.14(a) depicts the response time when replaying hours 120-122 of the MSR workload, after a the algorithms have gathered a 5 five day knowledge of the workload and have reach a steady state for a 200 GB SSD. We see a couple of interesting behaviors. Most notably it appears the size of the chunk has a notable impact on tiering even once the long term pattern have been captured and the data placed in its appropriate tier. When comparing the response of tiering and caching we see that tiering has similar performance to write through caching. Write back exhibits an average 25% better performance than tiering.

When the systems are faced with changes in the workload we see a different behavior. Figure 6.14(b) depicts a workload experiencing a change in its working set, which corresponds to replaying hours 6-8 of the MSR workload driven by great amounts of data being read semi-sequentially. We see that the best performing tiering configuration (1 MB) achieves comparable performance to the best performing caching counterpart. This might seem a bit surprising, since caching systems are designed to be quick to adapt to workload changes. However, further analysis revealed that tiering benefited from a combination of two factors, the throttling detection

mechanism employed by tiering quickly correcting performance degradation and the fact that tiering uses out of band data migrations. Nevertheless, we believe the data migration scheme in caching could be improved. Currently on every cache miss a the system waits for a 1 MB read I/O to HDD to complete, before returning to the user. If we issued only the I/O being requested to the HDD, returned to the user, and allow the migration to occur out of band we could reduce potentially reduce latency by a noticeable margin.

6.4.3 Leveraging Device Heterogeneity

Typical caching techniques can degrade performance by oversubscribing the SSD, while aiming to provide a higher hit rate [WR10]. This is due to a failure to balance the load among the available devices in an effective manner. Tiering solutions, on the other hand, are built precisely to leverage heterogeneous cost-performance-power characteristics across many device types. In the EDT work described earlier in this thesis, we proposed a three tier systems composed of SSDs, SAS, and SATA disks and argue that each device type has a place in the storage hierarchy for enterprise workloads [GPG+11]. Particularly, we showed that each tier is optimal with respect to one of the three metrics ($/GB, $/MB/s, and $/IOPS) and that sequential data access is served more *cost effectively* using either SAS or SATA drives depending on the I/O intensity of the sequential access. We have revisited the numbers and recalculated them according device prices as of Jun 2012. Table 6.4 depicts the device prices according to cost and the other three metrics presented in [GPG+11]. The most significant change comes in the SSD devices, where the price has decreased by more than half, making this device type also attractive in terms of $/Mb/s as well as $/IOPS.

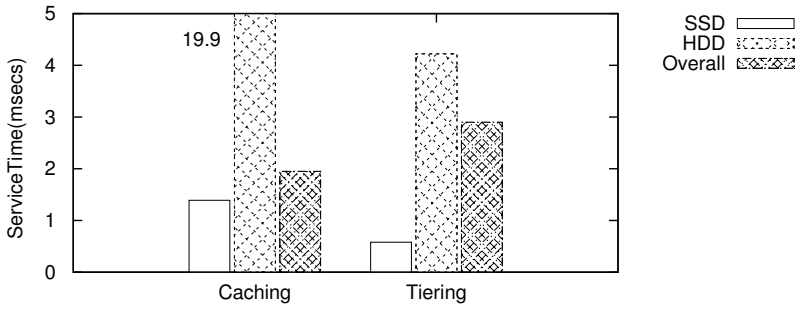


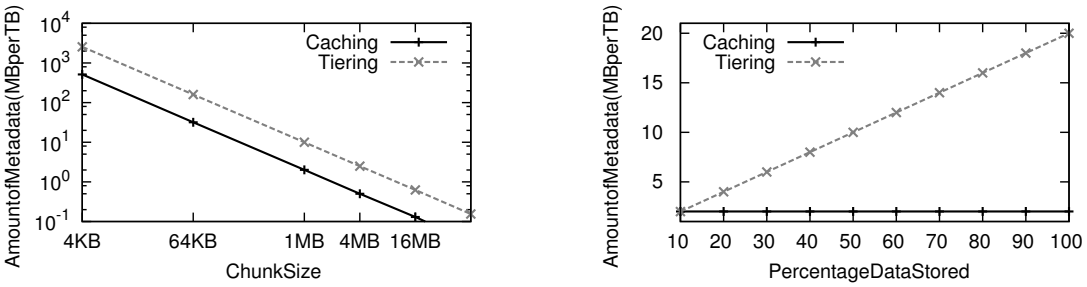
Figure 6.15: Average service time per device for the Caching and Tiering systems for the 120-122 hour period.

The distinction with respect to leveraging device heterogeneity seems central to the caching vs. tiering comparison. Figure 6.15 depicts the average response times for the replay of hours 120-122 of the MSR workloads. Notice that caching is able to achieve better overall response time, even though both its averages for SSD and HDD are worst that those of tiering. This is because the caching solution only sends 1 of every 33 I/Os to the HDD but incurring in a very high penalty for those I/Os. Tiering is being somewhat more proportional about half of the load to the SSD. However, since it is also taking into account other workload characteristics the its placement decisions appear to be yield a balanced load among devices. In the future, we envision that researchers will look into considering additional workload characteristics in caching algorithms, such as access pattern and I/O type, to leverage the benefits of distinct device types.

6.4.4 Metadata Overhead

Both caching and tiering solutions incur metadata overhead. Access to such metadata must be quick and therefore it is usually kept in fast, but limited, DRAM memory. The metadata maintained by caching is largely statistics related to lines in the cache, and ghost buffers which are typically few times bigger than the size

of cache. Thus, metadata requirement is a factor of the size of the cache and the amount of metadata per cache line, that is **O(cache size × line size)**. Tiering on the other hand stores information about each chunk that is accessed. Over time, this could be close to the total size of the back-end store, **O(total store size × chunk size)**. To reduce metadata overhead, chunk sizes are typically large, of the order of few megabytes. Figure 6.16 provides a comparison of metadata overhead for the two approaches. We see that tiering could use up to 10X the amount of metadata as caching depending on how much space is used.



(a) Varying the chunk size

(b) Varying the amount of data stored

Figure 6.16: Metadata overheads for the two techniques. The calculations assume the SSD is 10% the total size of the data; 3x ghost buffer size, with 8 byte metadata per chunk for caching. For tiering we assume 20 bytes of metadata per chunk. (a) Assumes that 50% chunks are never accessed, (b) Assumes chunk size of 64 KB.

In the future, we envision that chunk sizes in tiering would become smaller, while the metadata overhead is mitigated by storing it on the SSD and caching only a subset of it in DRAM. Given that there is locality in accesses to data, the same locality can also be exploited to cache chunk metadata in memory efficiently. Alternate optimizations are also possible. Chunks that do not get accessed frequently can be collapsed into a single piece of metadata. The effectiveness of these optimizations is yet to be seen and we expect research community to try out these and other similar solutions that exploit workload characteristics and reduce the overhead for tiering based solutions.

6.4.5 Reliability

We evaluate the reliability implications of caching and tiering using the shared SSD architecture *shared* (Figure 6.3(a)). In this case exclusive tiering is an attractive option, especially if the tiering implementation can exploit workload as well as device characteristics for best performance. Caching can also be employed with the additional ability to use the SSD as a read as well as persistent write cache. Finally, if one is using a reliable SSD layer, it is more cost-effective to use it as an exclusive layer instead of inclusive.

Apart from the previously mentioned reliability concerns, both caching and multi-tier systems depend on a mapping of data to devices. Mappings are updated after data is moved and it is crucial that this mapping be always consistent and durable. Caching, will typically perform map updates when bringing data to the cache. Thus, if the hit ratio is low and many new lines are being brought into the cache, we would be required to provide persistence in the order on tens or hundreds of transactions per second only for the keeping an updated copy of the mapping on a persistent medium. Tiering will update the map when data is being moved, which typically occurs in batches in the scale of minutes. Hence, enforcing less pressure to maintain an updated persistent version of the mapping.

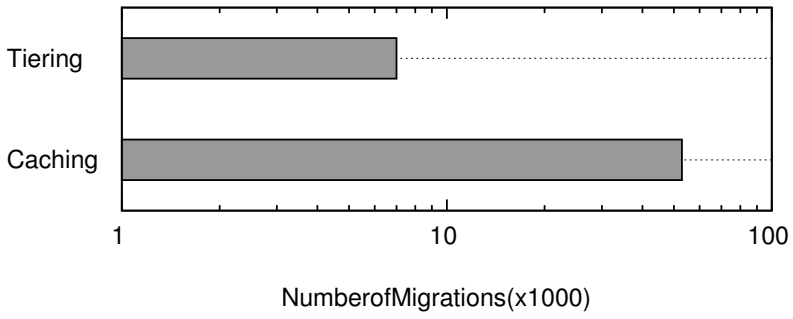


Figure 6.17: Amount of 1 MB data movements between SSD and HDD incurred by caching and multi-tiered systems during the reply of hours 120-122.

To validate the above hypothesis that tiering incurs in less data movements between the SSD and HDD tiers (or migrations) than caching we look at the number of migrations performed by both systems during hours 120-122. Figure 6.17 shows that tiering does less migrations almost an order of magnitude less migrations than the caching systems, even though the caching system achieves more than 95% hit ratio for the observed period. This leads us to think that designers of caching systems need to be very cautious handling updates to the persistent mappings, more so if using write back caching where there is only one update-to-date copy of the chunk. Particular attention must be given to misses on a write, given that it will trigger both a write to the SSD and an update on the map and both most occur simultaneously to ensure correctness and avoid losing data. This however is a hard task and current systems lack mechanisms that provide persistence of data structures (like maps), while guaranteeing consistency of their updates.

6.5 Discussion and Summary

Heterogeneous storage systems composed of a mix of SSD and HDD devices are becoming increasingly popular. Two alternate designs are caching and multi-tiering. Both systems take quite distinct approaches to managing distinct types of storage devices. We found that each of them have strengths and weaknesses.

We have to start by pointing out that our implementations of either caching and tiering systems by no means represent solutions available in the market. Many improvements can be made to both implementations which could potentially increase their performance significantly. Our focus in this study was to understand the differences between to two approaches and how each operates.

Our most important finding is that caching implementation continuously outperformed the tiering counterpart. In summary, caching adapts quicker to workload

changes while using less metadata. Nevertheless, caching systems need to handle more reliability concerns that tiering.

One of our most unexpected findings was the fact that bigger cache lines lead to increase hit rate and improved performance. We found this due mainly to the spatial locality within the workload, indicating a high probability of subsequent I/Os being within a 1 MB region of the previous. We found this not to be true for tiering, in which case the smallest chunk size tested (1 MB) provided the best performance, at a significant metadata overhead penalty though. On the other side, one potential advantage that tiering can have over caching is the ability to load balance among the different device types. However, we believe the performance gap between SSD and HDD will continue to widen, with SSDs becoming increasingly faster and cheaper. This would play down the need for load balancing. Also, since tiering approaches maintain metadata about every chunk ever accessed they might be able to decide on a better placement of data. Nevertheless, we did not evidence this to be true for the workloads we tested. We feel that more advanced chunk placement algorithms yield better results by further exploiting the load balancing among the different arrays.

Finally, both approaches present challenges in maintaining persistent metadata. For multi-tiering the challenge is keeping track of metadata to account for all data ever accessed; for caching is the high amount of updates in the cache triggered by moving in and out of the cache. Thus, in each case developers face significant complications making system metadata persistent. In the next chapter we address present a system that frees developers from to manually taking care of such metadata.

CHAPTER 7

MAKING EXTENSION DATA PERSISTENT

Previously in chapter 4 we presented three self-managed storage extensions. We concluded this section with the lessons learned while developing these systems. Among those lessons were the need for an easier method for developers to create self-managed storage extensions including providing support to create persistent data structures. As and indication of the importance of such support, all the extensions that we have discussed in this thesis thus far require persistence for some of their in-memory data structures. In this chapter, we present the design and implementation of a system that enables this much needed feature.

Operating systems offer developers two abstractions for managing data. A *malloc* style abstraction allows developers to flexibly allocate volatile memory for computation, and a *file* abstraction which provides data persistence. To make data persistent, developers write in-memory structures to non-volatile storage; likewise, they read such information into memory before use. The above practice is not ideal for development. Developers must carefully track persistent data structures in their code and ensure the atomicity of persistent modifications to related data structures. Developers are also required to implement serialization/deserialization for their structures which implies creating and managing additional metadata whose modifications must also be made consistent with the data they represent. These requirements increase code complexity, reduce code reliability and maintainability, and prolong the development time. Additionally, since processes directly interact with the storage system, the developer must be acutely aware of storage system operation to optimize the I/O. Given the diversity of storage and file systems, any storage-specific optimizations can hardly be universal.

Software Persistent Memory (SoftPM), is a lightweight abstraction for persistent memory. Unlike solutions in the literature, SoftPM provides a novel form of *orthogonal persistence* [Atk78], whereby the persistence of data (the *how*) is seamless to the developer, while providing a substantial degree of control over *when* and *what* data persists. SoftPM enables developers to allocate and interact with persistent memory in much the same way as they do with volatile memory. To use SoftPM, developers create one or more persistent *containers* to house a subset of in-memory data that they wish to make persistent and define container *root structures*. The developer only needs to ensure that a container's root structure house pointers to the data structures they wish to make persistent (e.g. head of a list or root of a tree). SoftPM automatically discovers new data reachable from a container's root structure (by recursively following pointers) and makes all new and modified data persistent when the application specifies a *persistence point*. Restoring a container returns the container root structure from which all originally reachable data can be accessed. SoftPM thus obviates the need for explicitly managing persistent data and places no restrictions on persistent data size or location(s) in the process' address space, while itself optimizing for I/Os via modular, back-end specific I/O drivers. Next, we discuss the the design and architecture of SoftPM

7.1 Persistence of Memory

In this section we identify design goals for a persistent memory system, examine the literature, and discuss how SoftPM differs from existing solutions.

7.1.1 Design Goals

Flexibility: Persistence is often necessary for only a fraction of the address space. – e.g.,10-50% in high-end computing (HEC) applications [OAT+07] and much lesser

Container Root Struct	Usage

`struct c_root {`
`  list_t *l;`
`} *cr;`
	`id = pCAlloc(m,sizeof(*cr),&cr);`
	`cr->l = list_head;`
	`cv = pPoint(id);`

Figure 7.1: Implementing a persistent list.

for metadata persistence in systems software [FB04, GPK+07, KR10, LTSY+07, LCSZ04, MKM+10, MAC+08, MG03, QD02, RCP08, SWS05, ZLP08]. By providing developers the flexibility to control what data must be made persistent, the minimal set of data needed to recover state can be chosen to reduce the size of persistence points.

Usability: Providing developers the necessary flexibility can introduce complexity. Candidate solutions may require developers to write additional code to track allocations and modifications to persistent data and/or explicitly map persistent data into immovable segments. An easy to use interface that automates much of the above, represents a near zero learning curve, and places minimal restrictions on persistent memory locations is desirable.

Performance: Creating persistence points is resource intensive and solutions that optimize I/O better are more relevant. Further, persistence solutions that minimize blocking of the application can directly lead to application visible performance gains.

Portability: The I/O characteristics of storage back-ends can be quite different. Specific storage back end optimizations may need to be substantially modified to for different back-ends. Portable persistence solutions are more desirable to developers.

7.2 SoftPM Overview

SoftPM implements a persistent memory abstraction called *container*. To use this abstraction, applications create one or more containers and associate a *root structure* with each. When the application requests a persistence point, SoftPM calculates a

Function	Description
`int pCAlloc(int magic,int cSSize,void **cStruct)`	*create a new container; returns a container identifier*
`int pCSetAttr(int cID, struct cattr *attr)`	*set container attributes; reports success or failure*
`struct cattr *pCGetAttr(int magic)`	*get attributes of an existing container; returns container attributes*
`void pPoint(int cID)`	*create a persistence point asynchronously*
`int pSync(int cID)`	*sync-commit outstanding persistence point I/Os; reports success or failure*
`int pCRestore(int magic, void **cStruct)`	*restore a container; populates container struct, returns a container identifier*
`void pCFree(int cID)`	*free all in-memory container data*
`void pCDelete(int magic)`	*delete on-disk and in-memory container data*
`void pExclude(int cID, void *ptr)`	*do not follow pointer during container discovery*

Table 7.1: The SoftPM application programmer interface.

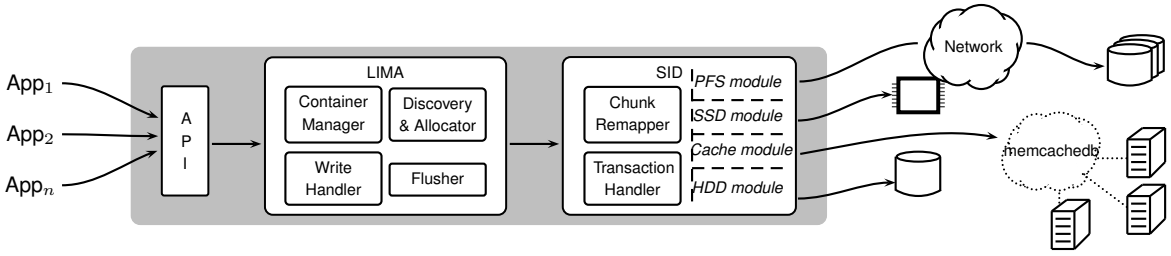


Figure 7.2: The SoftPM architecture.

memory closure that contains all data reachable (recursively via pointers) from the container root and writes it to storage atomically and (optionally) asynchronously.

The container root structure serves two purposes: *(i)* it frees developers from the burden of explicitly tracking persistent memory areas, and *(ii)* it provides a simple mechanism for accessing all persistent memory data after a restore operation. Table 7.1 summarizes the SoftPM API. In the simplest case, an application would create one container and create *persistence points* as necessary (Figure 7.1). Upon recovery, a pointer to a valid container root structure is returned.

7.2.1 System Architecture

The SoftPM API is implemented by two components: the *Location Independent Memory Allocator* (LIMA), and the *Storage-optimized I/O Driver* (SID) as depicted in Figure 7.2. LIMA's *container manager* handles container creation. LIMA manages the container's persistent data as a collection of memory pages *marked* for persistence. When creating a persistence point, the *discovery and allocator module* moves any data newly made reachable from the container root structure and located in volatile memory to these pages. Updates to these pages are tracked by the *write handler* at the granularity of multi-page *chunks*. When requested to do so, the *flusher* creates persistence points and sends the dirty chunks to the SID layer in an asynchronous manner. Restore requests are translated into chunks requests for SID.

The SID layer atomically commits container data to persistent storage and tunes I/O operations to the underlying storage mechanism. LIMA's flusher first notifies the *transaction handler* of a new persistence point and submits dirty chunks to SID. The *chunk remapper* implements a novel I/O technique which uses the property that all container data is memory resident and trades writing additional data for reducing overall I/O latency. We designed and evaluated SID implementations for hard drive, SSD, and memcached back-ends.

7.3 LIMA Design

Persistent containers build a foundation to provide seamless memory persistence. Container data is managed within a contiguous container virtual address space, a self-describing unit capable of being migrated across systems and applications running on the same hardware architecture. The container virtual address space is

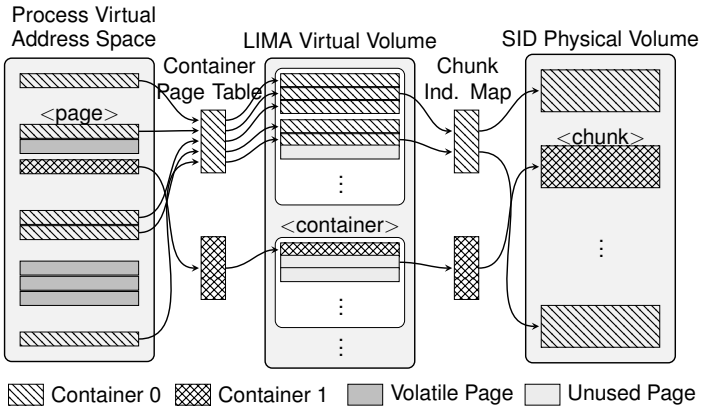
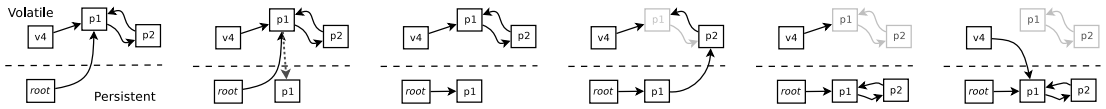


Figure 7.3: Container virtual address spaces in relation to process virtual address space and LIMA/SID volumes. The container virtual address space is chunked, containing a fixed number of pages (three in this case).

composed solely of pages marked for persistence including those containing application data and others used to store LIMA metadata. This virtual address space is mapped to logically contiguous locations within the virtual volume managed by LIMA. SID remaps LIMA virtual (storage) volumes at the chunk granularity to the physical (storage) volume it manages. This organization is shown in Figure 7.3. The indirection mechanism implemented by SID simplifies persistent storage management for LIMA which can use a logically contiguous store for each container.

7.3.1 Container Manager

The container manager implements container allocation and restoration. To allocate a new container (`pCAlloc`), an in-memory container page table, that manages both application persistent data and LIMA metadata, is first initialized. Next, the container root structure and other internal LIMA metadata structures are initialized to be managed via the container page table. To restore a container, an in-memory container instance is created and all container data and metadata loaded. Since container pages would likely be loaded into different portions of the process' address



(a) Start (b) Copy (c) Fix pointer (d) Add pointers (e) Repeat (f) Fix references

Figure 7.4: Container Discovery. Grey boxes indicate freed memory.

space, two classes of updates must be made to ensure consistency of the data. First, the container metadata must be updated to reflect the new in-memory data locations after the restore operation. Second, all memory pointers within data pages need to be updated to reflect the new memory addresses (pointer swizzling). To facilitate this, pointer locations are registered during process execution; we discuss automatic pointer detection in §7.5.

7.3.2 Discovery and Memory Allocation

A core feature of SoftPM is its ability to discover container data automatically. This allows substantial control over what data becomes persistent and frees the developer from the tedious and error-prone task of precisely specifying which portions of the address space must be allocated persistently. SoftPM implements automatic container discovery and persistent memory allocation by automatically detecting pointers in process memory, recursively moving data reachable from the container root to the container data pages, and fixing any *back references* (other pointers) to the data that was moved. In our implementation, this process is triggered each time a persistence point is requested by the application and is executed atomically by blocking all threads of a process only until the container discovery phase is completed; disk I/O is performed asynchronously (§7.3.4).

To make automatic container discovery possible, SoftPM uses static analysis and automatic source translation to register both *pointers* and *memory allocation*

requests (detailed in §7.5). At runtime, pointers are added either to a *persistent pointer set* or a *volatile pointer set* as appropriate, and information about all memory allocations is gathered. Before creating a persistence point, if a pointer in the persistent pointer set (except those excluded using `pExclude`) references memory outside the container data pages, the allocation containing the address being referenced is moved to the persistent memory region. Forward pointers contained within the moved data are recursively followed to similarly move other new reachable data using an edge-marking approach [KW99]. Finally, back references to all the data moved are updated. This process is shown in Figure 7.4. There are two special cases for when the target is not within a recognized allocation region. If it points to the code segment (e.g. function pointers), the memory mapped code is registered so that we can "fix" the pointer on restoration. Otherwise, the pointer metadata is marked so that its value is set to NULL when the container gets restored; this allows SoftPM to correctly handle pointers to OS state dependent objects such as files and sockets within standard libraries. If allocations made by library code are required to be persistent, then the libraries must also be statically translated using SoftPM; the programmer is provided with circumstantial information to help with this. In many cases, simply reinitializing the library upon restoration is sufficient, for instance, we added one extra line in SQLite (see § 7.6.3) for library re-initialization.

7.3.3 Write Handler

To minimize disk I/O, SoftPM commits only modified data during a persistence point. The *write handler* is responsible for tracking such changes. First, sets of contiguous pages in the container virtual address space are grouped into fixed-size *chunks*. At the beginning of a persistence point, all container data and metadata pages are marked *read-only*. If any of these pages are subsequently written into, two

alternatives arise when handing the fault: *(i)* there is no persistence point being created currently – in this case, we allow the write, mark the chunk *dirty*, and its pages *read-write*. This ensures at most one write page fault per chunk between two consecutive persistence points. *(ii)* there is a persistence point being created currently – then we check if the chunk has already been made persistent. If so, we simply proceed as in the first case. If it has not yet been made persistent, a copy of the chunk is first created to be written out as part of the ongoing persistence point, while write to the original chunk is handled as in the first case.

7.3.4 Flusher

Persistence points are created asynchronously (via `pPoint`) as follows. First, the flusher waits for previous persistence points for the same container to finish. It then temporarily suspends other threads of the process (if any) and marks all the pages of the container as read-only. If no chunks were modified since the previous persistence point, then no further action is taken. If modifications exist, the flusher spawns a new thread to handle the writing, sets the state of the container to *persistence point commit*, and returns to the caller after unblocking all threads. The handler thread first identifies all the dirty chunks within the container and issues write operations to SID. Once all the chunks are committed to the persistent store, SID notifies the flusher. The flusher then reverts the state of the container to indicate that persistence point has been committed.

7.4 SID Design

LIMA maps chunks and containers to its logical volume statically and writes out only the modified chunks during persistence points. If a mechanical disk drive is

used directly to store this logical volume, I/O operations during a persistence point can result in large seek and rotational delay overheads due to fragmented chunk writes within a single container; if multiple containers are in use simultaneously, the problem compounds causing disk head movement across multiple container boundaries. If a solid-state drive (SSD) were used as the persistent store, the LIMA volume layout will result in undesirable random writes to the SSD that is detrimental to both I/O performance and wear-leveling [GT05, KNM95]. The complementary requirement of ensuring atomicity of all chunk writes during a persistence point must be addressed as well. The SID component of SoftPM is an indirection layer below LIMA and addresses the above concerns.

7.4.1 SID Basics

SID divides the physical volume into chunk-sized units and maps chunks in the LIMA logical volume to physical volume locations for I/O optimization. The *chunk remapper* utilizes the property that all container data is memory resident and trades writing additional data (chunk granularity writes) for reducing I/O latency using device-specific optimizations.

Each physical volume stores *volume-level SID metadata* at a fixed location. This metadata includes for each container the address of a single physical chunk which stores two of the most recent versions of metadata for the container to aid crash recovery (elaborated later). To support chunk indirection,SID maintains a *chunk indirection map* as part of the container metadata. Finally, SID also maintains both an in-memory and on-disk per-container *free chunk bitmap* to locate the chunks utilized by a container. We chose to store per-container free chunk bitmaps to make each container self-describing and as a simple measure to eliminate race conditions when persisting multiple containers simultaneously.

During SID initialization, the free chunk bitmaps for each container stored on the physical volume are read into memory. An in-memory *global* free chunk bitmap obtained by merging the per-container free chunk bitmaps is used to locate free chunks in the physical volume quickly during runtime.

Atomic Persistence. To ensure atomicity of all chunk writes within a persistence point, SID uses persistence *version* numbers. When SID receives a request to create a persistence point, it goes through several steps in sequence. First, it writes all the dirty data chunks; chunks are never updated in place to allow recovery of the previous version of the chunks in case the persistence operation cannot be completed. Once the data chunk writes have all been acknowledged, SID writes the updated free chunk bitmap. Finally, it writes the container's metadata. This metadata includes, the chunk indirection map, the location of the newly written free chunk bitmap, and a (monotonically increasing) version number to uniquely identify the persistence point. Writing the last block of the metadata (the version number) after an I/O barrier commits the persistence point to storage; we reasonably assume that this block gets written to the storage device atomically.

Recovery. SID recovers the same way after both normal shutdowns and crashes. In either case, it identifies the most recent metadata for each container by inspecting their version numbers. It then reads the per-container free chunk bitmaps, and builds the global free chunk bitmap by merging all per-container bitmaps. When the application requests to restore a container, the most recent version of the chunk indirection map is used to reconstruct the container data in memory.

7.4.2 Device-specific optimizations

Disk Drives. Since sequential access to disk drives is orders of magnitude more efficient than random, we designed a mechanical disk SID driver to employ mostly-sequential chunk layout. The design assumes that the storage device will be performance rather than capacity bound, justifying a fair degree of space over-provisioning for the SID physical volume. Every chunk is written to the nearest free location succeeding the previously written location, wrapping around in a circular fashion. The greater the over-provisioning of the SID physical volume, the higher the probability of finding an adjacent free chunk. For instance, a 1.5X over-provisioning of capacity will result in every third chunk being free on average. Given sufficient outstanding chunk requests in the disk queue at any time, chunks can be written with virtually no seek overhead and minimum rotational delay. Reclaiming free space is vastly simpler than a log-structured design [RO91] or that of other copy-on-write systems like WAFL [HLM94] because *(i)* the design is not strictly log-structured and does not require multiple chunk writes to be sequential, and *(ii)* reclaiming obsolete chunks is as simple as updating a single bit in the free space bitmap without the need for log cleaning or garbage collection that can affect performance.

Flash drives. An SSD's logical address space is organized into *erase units* which were hundreds of kilobytes to a few megabytes in size for the SSD units we tested. If entire erase units are written sequentially, free space can be garbage collected using inexpensive *switch merge* operations rather than more expensive *full merge* operations that require data copying [KNM95]. SID writes to the SSD space one erase unit at a time by tuning its chunk size to a multiple of the erase unit size. The trade-off between the availability of free chunks and additional capacity provisioning follows the same arguments as those for disk drives above.

7.5 Pointer Detection

As discussed in §7.3, LIMA must track pointers in memory for automatic container discovery and updating pointer values during container restoration. The life-cycle of a pointer can be defined using the following stages: *(i) allocation*: when memory to store the pointer is allocated, *(ii) initialization*: when the value of the pointer is initialized, *(iii) use*: when the pointer value is read or written, and *(iv) deallocation*: when the memory used to store the pointer is freed. Note that, a pointer is always associated with an allocation. In SoftPM, we detect pointers at *initialization*, both explicitly (via assignment) or implicitly (via memory copying or reallocation). Hence, if programs make use of user-defined memory management mechanisms (e.g., allocation, deallocation, and copy), these must be registered with SoftPM to be correctly accounted for.

SoftPM's pointer detection works in two phases. At compile time, a static analyzer based on CIL (C Intermediate Language) [NMRW02] parses the program's code looking for instructions that allocate memory or initialize pointers. When such instructions are found, the analyzer inserts static hints so that these operations are registered by the SoftPM runtime. At runtime, SoftPM maintains an *allocation table* with one entry per active memory allocation. Each entry contains the address of the allocation in the process' address-space, size, and a list of pointers within the allocation. Pointers are added to this list upon initialization which can be done either *explicitly* or *implicitly*. A pointer can be initialized explicitly when it appears as an *l-value* of an assignment statement. Second, during memory copying or moving, any initialized pointers present in the source address range are also considered as implicitly initialized in the destination address range. Additionally, the source allocation

and pointers are deregistered on memory moves. When memory gets deallocated, the entry is deleted from the allocation table and its pointers deregistered.

Notes. Since SoftPM relies on static type information to detect pointers, it cannot record integers that may be (cast and) used as pointers by itself. However, developers can insert static hints to the SoftPM runtime about the presence of additional "intentionally mistyped" pointers to handle such oddities. Additionally, SoftPM is agnostic to the application's semantics and it is not intended to detect arbitrary memory errors. However, SoftPM itself is immune to most invalid states. For example, SoftPM checks whether a pointer's target is a valid region as per the memory allocation table before following it when computing the memory closure during container discovery. This safeguard avoids bloating the memory closure due to "rogue" pointers. We discuss this further detail in § 6.5.

Related work. Pointer detection is an integral part of garbage collectors [Wil92]. However, for languages that are not strongly typed, conservative pointer detection is used [BW88]. This approach is unsuitable for SoftPM since it is necessary to swizzle pointers. To the best of our knowledge, the static-dynamic hybrid approach to *exact* pointer detection presented in this paper, is the first of its kind. Finally, although pointer detection seems similar to *points-to* analysis [Hin01], these are quite different in scope. The former is concerned about if a given memory *location* contains a valid memory address, while the latter is concerned about *exactly which* memory addresses a memory location can contain.

7.6 Evaluation

Our evaluation seeks to address the correctness, ease of use, and performance implications of using SoftPM. We compare SoftPM to conventional solutions for persistence using a variety of different application benchmarks and microbenchmarks. In

cases where the application had a built-in persistence routine, we compared SoftPM against it using the application's default configuration. Where such an implementation was not available, we used the TPL serialization library [tpl] v1.5 to implement the serialization of data structures. All experiments were done on one or more 4-Core AMD Opteron 1381 with 8 GB of RAM using WDC WD5002ABYS SATA and MTRON 64GB SSD drives running Linux 2.6.31.

7.6.1 Workloads

We discuss workloads that are used in the rest of this evaluation and how we validated the consistency of persistent containers stored using SoftPM in each case.

Data Structures. For our initial set of experiments we used the DragonFly BSD [dra] v2.13.0 implementation of commonly used data structures including arrays, lists, trees, and hashtables. We populated these with large number of entries, queried, and modified them, creating persistence points after each operation.

Memcachedb [memb]. A persistent distributed cache based on memcached [mema] which uses Berkeley DB (BDB) [OBS99] v4.7.25 to persistently store elements of the cache. Memcachedb v1.2.0 stores its key-value pairs in a BDB database, which provides a native persistent key value store by using either a btree or a hash table. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB, and compared its performance to the native version using default configurations of the software. To use SoftPM with memcachedb, we modified the file which interfaced with BDB, reducing the LOC from 205 to 40. The workload consisted of inserting a large number of key-value pairs into memcachedb and performing a number of lookups, inserts, and deletes of random entries, creating persistence points after each operation.

SQLite [sql]. A popular serverless database system with more than 70K LOC. We modified it to use SoftPM for persistence and compared it against its own persistence routines. SQLite v3.7.5 uses a variety of complex data structures to optimize inserts and queries among other operations; it also implements and uses a custom slab-based memory allocator. A simple examination of the SQLite API revealed that all the database metadata and data is handled through one top-level data structure, called `db`. Thus, we created a container with just this structure and excluded an incorrectly detected pointer resulting from casting an `int` as a `void*`. In total, we added 9 LOC to make the database persistent using SoftPM which include a few more code to re-initialize a library.

MPI Matrix Multiplication. A recoverable parallel matrix multiplication that uses Open MPI v1.3.2 and checkpoints state across processes running on multiple machines.

7.6.2 Correctness Evaluation

To evaluate the correctness of SoftPM for each of the above applications, we crashed processes at random execution points and verified the integrity of the data when loaded from the SoftPM containers. We then compared what was restored from SoftPM to what was loaded from the native persistence method (e.g. BDB or file); in all cases, the contents were found to be equal. Finally, given that we were able to examine and correctly analyze complex applications such as SQLite with a large number of dynamically allocated structures, pointers, and a custom memory allocation implementation, we are confident that our static and dynamic analysis for pointer detection is sound.

Systems	Arrays	Lists	Hash Tables	Trees	O
BORG [BGU+09]	✓		✓	✓	✓
CDP [LTSY+07]				✓	
Clotho [FB04]	✓	✓			
EXCES [UGB+08]	✓	✓	✓	✓	
Deduplication [ZLP08]	✓		✓		
FlaZ [MKM+10]	✓	✓	✓		
Foundation [RCP08]	✓		✓		
GPAW [MHJ05]	✓				
I/O Shepherd [GPK+07]	✓			✓	
I/O Dedup [KR10]	✓			✓	✓
Venti [QD02]			✓		

Table 7.2: Persistent structures used in application and systems software. Arrays are multidimensional in some cases. *O* indicates other (e.g., graphs) and/or hybrid structures. In some cases, we assume an implementation based on design descriptions. This summary is created based on descriptions within respective articles and/or direct communication with the developers of these systems.

7.6.3 Case Studies

In this section, we perform several case studies including *(i)* a set of SoftPM-based persistent data structures, *(ii)* an alternate implementation of memcachedb [memb] which uses SoftPM for persistence, *(iii)* a persistent version of SQLite [sql], a serverless database based on SoftPM, and *(iv)* a recoverable parallel matrix multiplication application that uses MPI.

Making Data Structures Persistent

We examined several systems that require persistence of in-memory data and realized that these systems largely used well-known data structures to store their persistent data such as arrays, lists, trees, and hashtables. A summary of this information is presented in Table 7.2. We constructed several microbenchmarks that create and modify several types of data structures using SoftPM and TPL [tpl], a data structure serialization library. To quantify the reduction in development

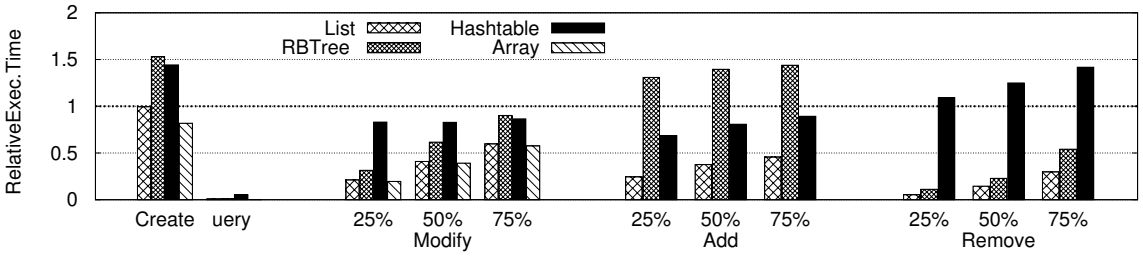


Figure 7.5: Performance of individual data structure operations. The bars represent the execution time of the SoftPM version relative to a version that uses the TPL serialization library for persistence. We used fixed size arrays which do not support add or remove operations.

Data Structure	**Original LOC**	**LOC for Persistence**	**LOC to use SoftPM**
Array	102	17	3
Linked List	188	24	3
RB Tree	285	22	3
Hash Table	396	21	3
SQLite	73042	6696	9
memcachedb	1894	205	40

Table 7.3: Lines of code to make structures (or applications) persistent and recover them from disk. We used TPL for Array, Linked List, RB Tree, and Hash Table; SQLite and memcachedb implement custom persistence.

complexity we compared the lines of code necessary to implement persistence for various data structures using both solutions. We report in Table 7.3 the lines of code (LOC) without any persistence and the additional LOC when implementing persistence using TPL and SoftPM respectively.

For each data structure we perform several operations (e.g modify) and make the data structure persistent. Note that the TPL version writes entire structures to disk, whereas SoftPM writes only what was modified. For *create*, SoftPM calculates the memory closure, move the discovered data to persistent memory, and write to disk and overhead is proportional to this work. The *query* operation doesn't modify any data and SoftPM clearly outperforms TPL in this case. *modify* only changes existing data values, *remove* reduces the amount of data written by TPL and involves only metadata updates in SoftPM, and *add* increases the size of the data structure

increasing both the amount of data and metadata writes with SoftPM. Figure 7.5 presents the execution times of the SoftPM version relative to the TPL version. Two interesting points are evidenced here. First, for add operations SoftPM outperforms TPL for all data structures except RB Tree, this is due to balancing of the tree modifying almost the entire data structure in the process requiring expensive re-discovery, data movement, and writing. Second, the remove operations for Hashtable are expensive for SoftPM since its implementation uses the largest number of pointer; removing involves a linear search in one of our internal data structures and we are currently working on optimizing this.

Comparing with Berkeley DB

memcachedb is an implementation of memcached which periodically makes the key value store persistent by writing to a Berkeley DB (BDB) [OBS99] database. BDB provides a persistent key value store using a btree (BDB-Btree) or hash table (BDB-HT), as well as incremental persistence by writing only dirty objects, either synchronously or asynchronously. We modified memcachedb to use a hash table which we make persistent using SoftPM instead of using BDB. In Figure 7.6 we compare the operations per second achieved while changing the persistence back-end. SoftPM outperforms both variants of BDB by upto 2.5X for the asynchronous versions and by 10X for the synchronous.

Making an in Memory Database Persistent

SQLite is a production-quality highly optimized serverless database, it is embedded within many popular software such as Firefox, iOS, Solaris, and PHP. We implemented a benchmark which creates a database and performs random insert, select, update, and delete transactions. We compare the native SQLite persistence to that

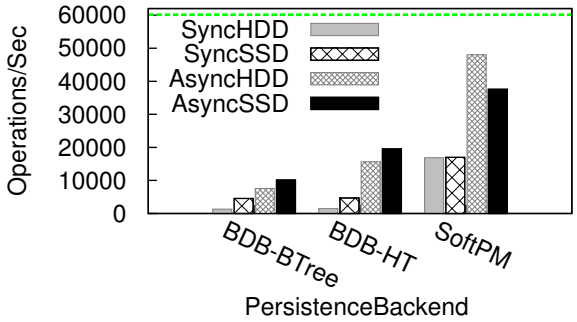


Figure 7.6: Performance of memcachedb using different persistent back-ends. The workload randomly adds, queries, and deletes 512 byte elements with 16 byte keys. The dashed line represents a memory only solution.

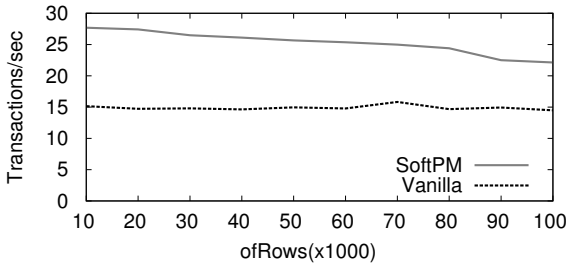


Figure 7.7: SQLite transactions per second comparison when using SoftPM and the native file persistence.

using SoftPM; transactions are synchronous in both cases. Figure 7.7 shows that SoftPM is able to achieve 55% to 83% higher transactions rate depending on the size of the database. We believe this is a significant achievement for SoftPM given two facts. First, SQLite is a large and complex code base which includes a complete stand alone database application and second, SQLite's file transactions are heavily optimized and account for more than 6K LOC. Further analysis revealed that most of SoftPM's savings arise from its ability to optimize I/O operations relative to SQLite. The reduction in performance improvement with a larger number of rows in the database is largely attributable to a sub-optimal container discovery implementation; by implementing incremental discovery to include only those pointers within dirty pages, we expect to scale performance better with database size in future versions of SoftPM. Figure 7.8 shows a breakdown of the total overhead in-

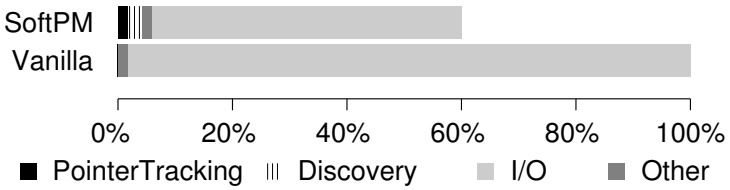


Figure 7.8: Breakdown of time spent in the SQLite benchmark for 100K rows.

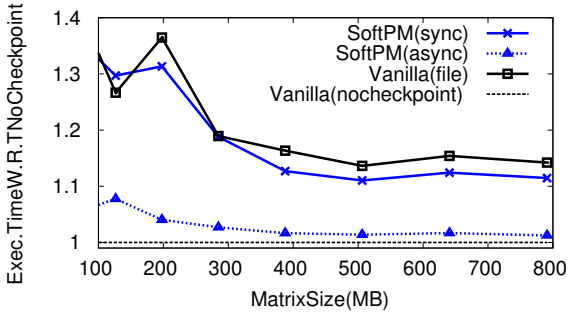


Figure 7.9: Contrasting application execution times for MPI matrix multiplication using 9 processes.

cluding I/O time incurred by SoftPM which are smaller than the time taken by the native version of SQLite. Finally, all of this improvement was obtained with only 9 additional LOC within SQLite to use SoftPM, a significant reduction relative to its native persistence implementation (6696 LOC).

Recoverable Parallel Matrix Multiplication

To compare SoftPM's performance to conventional checkpointing methods, we implemented a parallel matrix multiplication application using Cannon's algorithm [GS94]. We evaluated multiple solutions, including a no checkpoint non-recoverable implementation, a serialization-based implementation which serializes the matrices to files, and *sync* and *async* versions of SoftPM, in all cases a checkpoint is made after calculating each sub-matrix. For the file-based checkpointing version we added 79 LOC to serialize, write the matrix to a file, and recover from the file. In the SoftPM

version, we added 44 LOC, half of them for synchronization across processes to make sure all processes restored the same version after a crash.

Figure 7.9 compares the total execution time across these solutions. Synchronous SoftPM and the serialization solution have similar performance. Interestingly, because of unique ability of overlapping checkpoints with computation, the asynchronous version of SoftPM performs significantly better than either of the above, in fact, within a 1% difference (for large matrices) relative to the memory-only solution.

7.6.4 Microbenchmarks

In this section, we evaluate the sensitivity of SoftPM performance to its configuration parameters using a series of microbenchmarks. For these experiments, we used a persistent linked list as the in-memory data structure. Where discussed, *SoftPM* represents a version which uses a SoftPM container for persistence; TPL represents an alternate implementation using the TPL serialization library. Each result is averaged over 10 runs, and except when studying its impact, the size of a chunk in the SID layer is set to 512KB. To make the linked list persistent, SoftPM and TPL add 5 and 28 LOC, respectively.

Incremental Persistence. Usually, applications modify only a subset of the in-memory data between persistence points. SoftPM implements incremental persistence by writing only the modified chunks, which we evaluated by varying the locality of updates to a persistent linked list, shown in Figure 7.10. As expected, TPL requires approximately the same amount of time regardless of how much data is modified; it always writes the entire data structure. The SoftPM version requires less time to create persistence points as update locality increases.

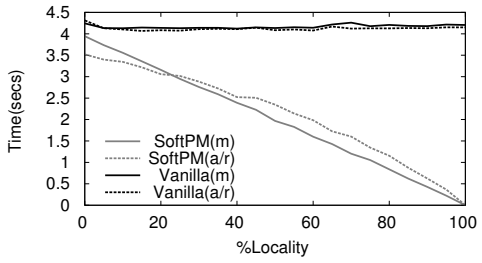


Figure 7.10: Impact of locality on incremental persistence. Two different sets of experiments are performed: *(m)* where only the contents of the nodes are modified, and *(a/r)* where nodes are added and removed from the list. In both cases the size of the list is always 500MB.

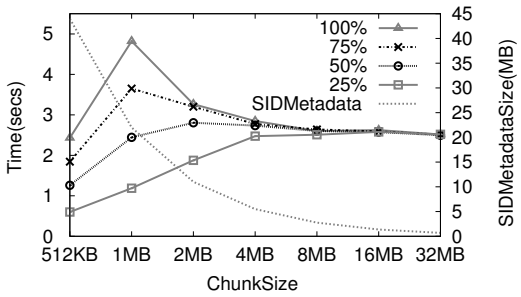


Figure 7.11: Impact of chunk size on persistence point time. A list of size 500MB is made persistent and individual lines depict for a specific fraction of the list modified.

Chunk Size. SoftPM tracks changes and writes container data at the granularity of a chunk to create persistence points. When locality is high, smaller chunks lead to lesser data written but greater SID metadata overhead because of a bigger chunk indirection map and free chunk bitmap. On the other hand, larger chunks imply more data written but less SID metadata. Figure 7.11 shows the time taken to create persistence points and the size of the SID metadata at different chunk sizes.

Parallel Persistence Points. The SID layer optimizes the way writes are performed to the underlying store, e.g. writing to disk drives semi-sequentially. Figure 7.12 depicts the performance of SoftPM in relation to TPL when multiple processes create persistence points to different containers at the same time. We vary the number of processes, but keep the total amount of data persisted by all the processes a constant. The total time to persist using SoftPM is a constant given that the same

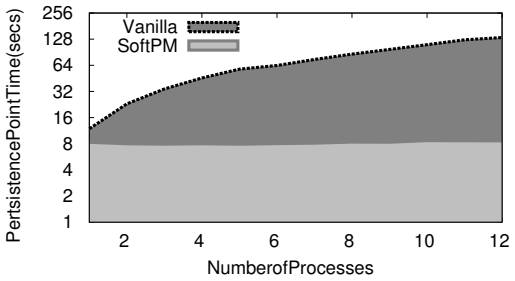


Figure 7.12: Time to create persistence points for multiple parallel processes. Every process persists a list of size (1GB/number-of-processes).

amount of data is written. On the other hand, the time for TPL increases with the number of threads, because of lack of optimization of the interleaving writes to the different container files at the storage level.

Percentage of Pointers in Data. Creating a persistence point requires computing a transitive memory closure, an operation whose time complexity is a function of the number of pointers in container data. We varied the fraction of the memory (used by the linked list) that is used to store pointers (quantified as "percentage pointers in data") and measured the time to create a full (non-incremental) persistence point.

We compare performance with a TPL version of the benchmark that writes only the contents of the elements of the list to a file in sequence without having to store pointers. A linked list of total size 500MB was used. Figure 7.13 shows the persistence point creation times when varying the percentage pointers in data. SoftPM is not always more efficient in creating persistence points than TPL, due to the need to track and store all the pointers and the additional pointer data and SoftPM metadata that needs to be written to storage. The linked list represents one of the best case scenarios for the TPL version since the serialization of an entire linked list is very simple and performs very well due to sequential writing. We also point out here that we are measuring times for registering pointers in the entire list, a full discovery and (non-incremental) persistence, a likely worst case for SoftPM;

in practice, SoftPM will track pointers incrementally and persist incrementally as the list gets modified over time. Further, for the complex programs we studied the percentage pointers in data is significantly lower; in SQLite this ratio was 4.44% and for an MPI-based matrix multiplication this ratio was less than 0.04%. Finally, the amount of SoftPM metadata per pointer can be further optimized; instead of 64 bit pointer locations (as we currently do), we can store a single page address and multiple 16 bit offsets.

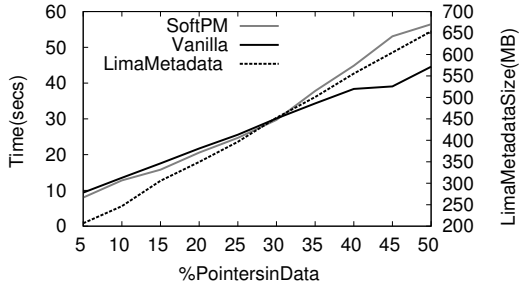


Figure 7.13: Time to persist a linked list and LIMA metadata size, varying percentage of pointers in data. The total size is fixed at 500MB and node sizes are varied accordingly.

7.7 Discussion

Several issues related to the assumptions, scope, and current limitations of SoftPM warrant further discussion and also give us direction for future work.

Programmer errors. SoftPM's automatic discovery of updated container data depends on the programmer having correctly defined pointers to the data. One concern might be that if the programmer incorrectly assigned a pointer value, that could result in corrupt data propagating to disk or losing portions of the container. This is a form of programmer error to which SoftPM seems more susceptible to. However, such programmer errors would also affect other classes of persistence solutions including those based on data structure serialization since these also require navigat-

ing hierarchies of structures. Nevertheless, SoftPM does provide a straightforward resolution when such errors get exercised. While not discussed in this paper, the version of SoftPM that was evaluated in this paper implements container versioning whereby previously committed un-corrupted versions of containers can be recovered when such errors are detected. Additionally, we are currently implementing simple checks to warn the developer of unexpected states which could be indicators of such errors; e.g., a persistent pointer points to a non-heap location.

Container sharing. Sharing container data across threads within a single address-space is supported in SoftPM. Threads sharing the container would have to synchronize updates as necessary using conventional locking mechanisms. Sharing memory data across containers within a single address-space is also supported in SoftPM. These containers can be independently checkpointed and each container would store a persistent copy of its data. However, sharing container data persistently is not supported. Further, in our current implementation, containers cannot be simultaneously shared across process address-spaces. In the future, such sharing can be facilitated by implementing the SoftPM interface as library system calls so that container operations can be centrally managed.

Non-trivial types. SoftPM currently does not handle pointers that are either untyped or ambiguously typed. This can occur if a programmer uses a *special* integer type to store a pointer value or if a pointer type is part of a union. These can be resolved in the future with additional hints to SoftPM's static translator from the programmer. Additionally, the runtime could hint to SoftPM about when a union type resolves to a pointer and when it is no longer so.

Unstructured data. The utility of SoftPM in simplifying development depends on the type of the data that must be made persistent. Unstructured data (e.g., audio or video streams) are largely byte streams and do not stand to benefit as much from

SoftPM as data that has structure containing a number of distinct elements and pointers between them. Specifically, unstructured data tends not to get modified in place as much as structured data and consequently they may not benefit from the incremental change tracking that SoftPM implements.

7.8 Summary

Storage system extensions, as well as many other classes of applications, rely on a portion of their state being persistent to ensure correctness for continued operation, the availability of a lightweight and simple solution for memory persistence becomes increasingly important. From our previous experiences building storage system extensions we witnessed the difficulties that arise to maintain extension metadata in a consistent state. With SoftPM, we believe that we have found a radical solution to provide memory persistence that is both practical and effective. Developers use existing memory interfaces as-is, needing only to instantiate persistent containers and container root structures besides requesting persistence points. SoftPM automates persistence by automatically discovering data that must be made persistent for correct recovery and ensures the atomic persistence of all modifications to the container. Recovery of persistent memory is equally simple with SoftPM returning a pointer to the container root via which the entire container can be accessed.

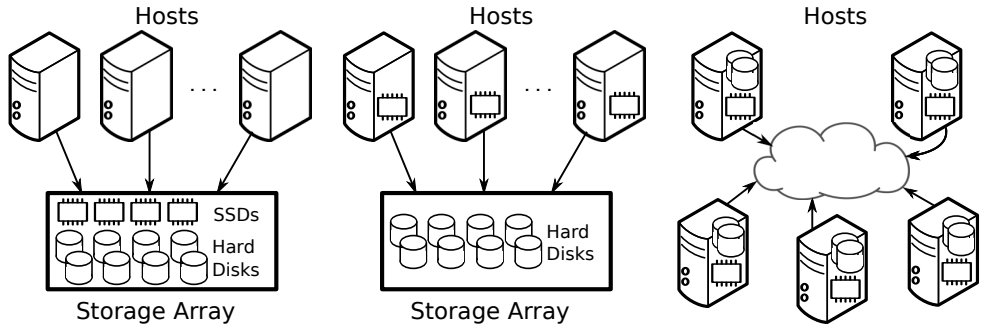
We implemented a prototype user-level version of SoftPM and evaluated it using a range of microbenchmarks, an MPI application, SQLite database, and a distributed memcachedb application. Development complexity as measured using lines of code was substantially reduced when using SoftPM relative to native application level persistence or using an off-the-shelf serialization library. Performance results were also very encouraging with improvements of up to 10X, with SoftPM's asynchronous persistence feature demonstrating the potential for performing at close to

memory speeds. An in-kernel library version for SoftPM will allow storage extensions to benefit from it as well. We believe this to be a worthwhile future direction.

CHAPTER 8

FUTURE WORK

In previous chapters we have seen how to improve both storage extension development and administration. Practically all of our research has focused on centralized storage, where all the storage devices are contained within a single box and access is provided through the network (see Figure 8.1(a)). These systems are currently the preferred storage architecture, given that they are stable, have an existing set of management tools and their behavior is well understood. However, there are also some important drawbacks of this approach. First, scalability; adding more storage will also demand careful planning from storage administrators to get the most benefits. Second, although these systems are very reliable they still present a single point of failure. Third, they are usually very expensive. Centralized storage solutions are widely available from multiple storage vendors like EMC, IBM, and NetApp, among others.



(a) Centralized (b) Mixed (c) Distributed

Figure 8.1: Different storage architectures

Over the past decade solid state devices have become ubiquitous in enterprise storage systems [Lal09, Tan10, Pet09b]. With this addition came significant performance and power savings. While these improvements have been substantial, researchers are noticing network access as a factor limiting latency improvements, and

that further improvements could be obtained by having the storage devices closer to the where data is accessed. This has led to a resurgence of interest in mixed and fully distributed storage architectures (depicted in Figures 8.1(b) and 8.1(c) respectively). These types of architecture offer many benefits over the centralized approach. First, they can substantially reduce access latency by eliminating most of the network accesses. Second, they promise better scalability; users can increase overall performance by simply adding more physical resources. Finally, they can reduce capital cost by providing smaller, more compact, building blocks.

We believe that the mixed and distributed architectures present unique opportunities for storage systems research as many challenges remain to be addressed. In particular we identify three research opportunities related to the content discussed in this dissertation. First, current administration tools for distributed architectures are very rudimentary in comparison to those available for centralized storage. One of the points in favor of distributed architectures is their ability to reduce latency. For this to be true, data must be physically close to where it is accessed. However, providing an effective solution for this is not trivial. Data access patterns can change through time making a previous data layout ineffective. Having storage administrators analyze such information can be overwhelming. Hence, we believe that tools that automatically optimize data placement depending on where it is accessed are a necessary first step for these systems to be successful.

Additionally, ensuring data reliability in distributed architectures is much more complicated. Not only can individual devices fail but also we additionally have to deal with host failures and network connectivity issues. Expanding ABLE to provide development tools for distributed environments will reduce developer effort and seems like a viable next improvement. These augmented development tools must allow developers to tune their systems by configuring their systems in terms of

reliability (e.g. number of data replicas to maintain), consistency (e.g. how frequent to write data in volatile memory to disk) and performance (e.g. when and how to update the replicas) guarantees.

Finally, maintaining metadata consistent is a particular hard task in distributed storage architectures. Our current techniques to handle persistent memory have to be adapted and potentially redesigned to be able to work in distributed settings. Among the issues that must be addressed are deciding when to persist the data. Should we persist changes in all hosts or only in a subset? If only in a subset, how do we ensure that changes will achieve a consistent state? Providing concrete answers to these questions, as well as the other issues mentioned above, involves a significant thought process and will prove valuable towards enabling efficient distributed architectures.

CHAPTER 9

CONCLUSIONS

Storage is one of the most rapidly growing systems in the data center since the pace at which we generate information and the number of users grows every year. Hence, storage administrators face a harder challenge to make it work efficiently. We believe that future storage systems will need to support multiple extensions that improve properties of the storage system such as performance, energy efficiency, or reliability. Most of the work in this dissertation has focused on the development, support, and correct deployment of self-managed storage extensions. From what we learned in this process we deduced the need for a storage development and deployment infrastructure, this lead to the creation of ABLE.

The current ABLE system has two main features. First, it providers developers with a well defined and commonly used block functionality, including the ability to provide memory persistence. Second, the extension stacking theory which aids administrators decide how to order sets of extensions to achieve high level system goals. However, as stated before, this theory has some limitations. For instance, if the administrator wants to improve storage system performance and has multiple extensions to choose from, which one should she deploy? In these cases the current stacking rules will not provide much insight to help the administrator since the decision needs to be based on the semantics of the extensions and the characteristics of the workload that the system observes.

To help administrator choose the more appropriate extension between those that accomplish the same goal, we provide an empirical methodology that considers both the internal workings of the extension and the characteristics of the workload it will be subject to. We developed and evaluated two independent storage extensions, *caching* and *multi-tiering*, and analyzed their behavior under a variety of scenarios to

determine when is either one more favorable. This enabled us to deduct meaningful observations about the effects of particular workload patterns on the extensions and infer which will perform the best under the workload studied.

Finally, when developing all of the storage extensions that were discussed in this thesis we faced challenges to make extension metadata persistent. We designed and developed software persistent memory (SoftPM) to provide a simple and easy to use interface for arbitrary data structures to be made persistent upon user request. Using a combination of static and runtime analysis SoftPM largely frees developers from creating the mechanisms necessary for making in-memory data structures persistent.

In conclusion, we believe this is a crucial time for storage systems. During much of the past decade we saw the performance between CPUs and disk-based storage widen. CPU performance improved following Moore's Law but disk-based storage stagnated only making improvements in capacity and sequential throughput. But, recently a mix factors have came together that are revolutionizing the ways in which we build storage. First, workload consolidation driven by the success of virtualization has substantially increased the demand for IOPS from storage systems. Second, the proliferation of solid state disk as a viable alternative to provide needed IOPS is gaining momentum. Third, the rate at which we generate and store data is increasing more than ever. These factors and other factors are forcing storage companies to quickly add functionality (in the form of extensions) to their systems. We believe this dissertation provides fundamental solutions that will ensure the timely adoption of these critical extensions in future storage systems.

BIBLIOGRAPHY

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. *Proc. of the Summer USENIX Conference*, June 1986.

[ABG+01] Guillermo Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[AHH+01] Eric Anderson, Joe Hall, Jason Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. *Lecture Notes in Computer Science*, 2141/2001:145–158, 2001.

[AHK+02] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. *USENIX Conference on File and Storage Technologies*, January 2002.

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

[ASS+05] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, 2005.

[ASV06] Marcos K. Aguilera, Susan Spence, and Alistair Veitch. Olive: Distributed Point-in-Time Branching Storage for Real Systems. *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, June 2006.

[Atk78] Malcolm P. Atkinson. Programming Languages and Databases. In *VLDB*, 1978.

[Ave10] Avere Systems, Inc. The Avere Architecture for Tiered NAS. White Paper, 2010.

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

[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.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the USENIX Security Symposium*, August 2003.

[BGU+09] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization and Self-optimization in Storage Systems. In *Proc. of USENIX FAST*, 2009.

[BMP+04] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level Checkpointing for Shared Memory Programs. *SIGARCH Comput. Archit. News*, 32(5):235–247, 2004.

[BP09] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proc. of NSDR*, 2009.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. *Proc. of the ACM Symposium on Operating Systems Principles*, pages 267–283, December 1995.

[BW88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience* , 18(9):807–921, September 1988.

[CAC+84] W. Paul Cockshott, Malcolm P. Atkinson, Kenneth J. Chisholm, Peter J. Bailey, and Ronald Morrison. POMS - A Persistent Object Management System. *Software Practice and Experience*, 14(1):49– 71, 1984.

[CCA+11] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of ASPLOS*, 2011.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of ACM SIGMOD*, 1994.

[CG02] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, 2002.

[CHIK03] Alva Couch, John Hart, Elizabeth G. Idhaw, and Dominic Kallas. Seeking Closure in an Open World: A Behavioral Agent Approach to Configuration Management. *Proc. of USENIX LISA*, pages 125–148, 2003.

[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.

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of SOSP*, 2009.

[Cou96] W. V. Courtright *et al.* RAIDframe: Rapid Prototyping for Disk Arrays. *SIGMETRICS Perform. Eval. Rev.*, 24(1):268–269, 1996.

[CPB03] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 86–97, 2003.

[Cus94] Helen Custer. Inside the Windows NT File System. *Microsoft Press*, August 1994.

[Dav93] Alan M. Davis. *Software Requirements: Objects, Functions, and States.* Prentice-Hall, Inc., 1993.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computer Systems*, 7(3):289–312, 1994.

[DGJ+05] Koustuv Dasgupta, Sugata Ghosal, Rohit Jain, Upendra Sharma, and Akshat Verma. Qosmig: Adaptive rate-controlled migration of bulk data in storage systems. In *Proc. of ICDE*, 2005.

[dIMO97] Miguel de Icaza, Ingo Molnar, and Gadi Oxman. Kernel korner: The new linux raid code. *Linux Journal*, 1997(44es), 1997.

[DKB95] Fred Douglis, P. Krishnan, and Brian N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[dra] DragonFlyBSD. http://www.dragonflybsd.org/.

[Eas08] Easy Computing Company. Managed Flash Technology. *http://www.easyco.com/mft/index.htm*, 2008.

[EP04] E. N. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE TDSC*, 1(2):97–108, 2004.

[Epp89] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, 1989.

[FB04] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proc. of IEEE MSST*, 2004.

[FB05] Michail D. Flouris and Angelos Bilas. Violin: A Framework For Extensible Block-level Storage. *Proc. IEEE Conference on Mass Storage Systems and Technologies*, April 2005.

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

[Gan03] G. R. Ganger *et al.* Self-* Storage: Brick-based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[Gar10] Gartner survey shows data growth as the largest data center infrastructure challenge. `http://www.gartner.com/it/page.jsp?id=1460213`, 2010.

[GAW09] Ajay Gulati, Irfan Ahmad, and Carl Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proc. of USENIX FAST*, February 2009.

[GPG+09] Jorge Guerra, Himabindu Pucha, Karan Gupta, Wendy Belluomini, and Joseph Glider. Energy Proportionality for Storage: Impact and Feasibility. In *Proc. of ACM/USENIX HotStorage*, 2009.

[GPG+11] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proc. of USENIX FAST*, February 2011.

[GPK+07] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proc. of ACM SOSP*, 2007.

[GPRA98] Doug Ghormley, David Petrou, Steven Rodrigues, and Thomas Anderson. SLIC: An Extensibility System for Commodity Operating Systems. *Proc. of the USENIX Annual Technical Conference*, June 1998.

[GR] Jorge Guerra and Raju Rangaswami. Stacking ABLE Extensions. *Available at: http://www.cs.fiu.edu/∼raju/reviewer/ABLE-Theory.pdf.*

[Gre08] Brendan Gregg. Zfs l2arc, July 2008.

[GS94] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. *Proc. of the ACM SPAA*, 1994.

[GS02] M.E. Gómez and V. Santonja. Characterizing Temporal Locality in I/O Workload. *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[GSJ+05] Roberto Gioiosa, José Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proc. of the ACM/IEEE SC*, 2005.

[GSKF00] 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 200.

[GT05] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[GV00] Garth A. Gibson and Rodney Van Meter. Network Attached Storage Architecture. *Communications of the ACM*, 43, Nov 2000.

[HD96] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO. *SGI Technical Report*, 1996.

[HD06] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Proc.of SciDAC Conference*, 2006.

[hfs04] HFS Plus Volume Format. *http://developer.apple.com/technotes/tn/tn1150.html*, 2004.

[HHS05] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proc. 20th ACM Symposium on Operating Systems Principles*, Oct 2005.

[Hin01] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proc. of the USENIX Technical Conference*, 1994.

[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.

[HSY05] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.

[IBM] IBM. IBM Attempts to Reinvent Memory. Online, http://www.technologyreview.com/ Nanotech/19477/page1/.

[IBM10] IBM Corporation. IBM System Storage DS8000 series. Data Sheet, 2010.

[Int98] Intel Corporation. Intel application launch accelerator. *Online at http://support.intel.com/support/chipsets/iaa/*, 1998.

[Int02] Intel Corporation. Intel®mobile platform vision guide for 2003, 2002.

[JLR+94] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proc. of VLDB*, 1994.

[JS94] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450. Morgan Kaufmann, 1994.

[JZ02] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Marina Del Rey*, pages 31–42. ACM Press, 2002.

[KBBA] Dave Kleikam, Dave Blaschke, Steve Best, and Barry Arndt. JFS for Linux. *http://jfs.sourceforge.net/*.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.

[KNM95] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-memory based File System. In *USENIX Technical*, 1995.

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

[KW99] Sheetal V. Kakkad and Paul R. Wilson. Address translation strategies in the texas persistent store. In *Proceedings of the USENIX Conference on Object-Oriented Technologies & Systems*, 1999.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of ACM SIGMOD*, 1996.

[Lal09] Bob Laliberte. Automate and Optimize a Tiered Storage Environment FAST! ESG White Paper, 2009.

[LCSZ04] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proc. of USENIX FAST*, 2004.

[Leh99] Greg Lehey. The Vinum Volume Manageer. *Proc. USENIX Technical Conference (FREENIX Track)*, June 1999.

[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.

[LKKK12] Sungjin Lee, Taejin Kim, Kyungho Kim, and Jihong Kim. Lifetime management of flash-based ssds using recovery-aware dynamic throttling. In *Proc. of USENIX FAST*, 2012.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Commun. ACM*, 34:50–63, October 1991.

[LNBZ08] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: trading address space for reliability and security. *SIGARCH Comput. Archit. News*, 36(1):115–124, 2008.

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

[LTSY+07] Guy Laden, Paula Ta-Shma, Eitan Yaffe, Michael Factor, and Shachar Fienblit. Architectures for controller based cdp. In *Proc. of USENIX FAST*, 2007.

[MAC+08] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. *Proc. of EuroSys*, 2008.

[Mar10] Howard Marks. Cache or Tier? Flash By Any Other Name... Howard Marks: Network Computing Blogger, March 2010.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[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.

[mema] memcached. http://memcached.org/.

[memb] memcachedb. http://memcachedb.org/.

[MG03] C.B. Morrey and D. Grunwald. Peabody: the time travelling disk. In *Proc. of IEEE MSST*, 2003.

[MGR03] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-Based Storage. *IEEE Communications Magazine*, August 2003.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970.

[MHJ05] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, Jan 2005.

[Mic06] Microsoft Corporation. Fast System Startup for PCs Running Windows XP. *Windows Platform Design Notes*, December 2006.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems 2*, 3:181–197, August 1984.

[MKM+10] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proc. of EuroSys*, 2010.

[MM03] Nimrod Megiddo and Dharmendra Modha. ARC: A self-tuning, low overhead replacement cache. In *In Proceedings of the 2003 Conference on File and Storage Technologies (FAST*, pages 115–130, 2003.

[Mos92] E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE TSE*, 18(8):657–673, 1992.

[MSSL97] Henry M. Mashburn, Mahadev Satyanarayanan, David Steere, and Yui W. Lee. RVM: Recoverable Virtual Memory, Release 1.3. *http://www.coda.cs.cmu.edu/doc/html/rvm_manual.html*, 1997.

[Nam] Namesys, Inc. The ReiserFS File System. *Go to: http://www.namesys.com/.*

[Nar08] D. Narayanan *et al.* Write Off-Loading: Practical Power Management for Enterprise Storage. *Proc. USENIX FAST*, Feb 2008.

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

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

[NDT+08] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. *Proc. of the USENIX OSDI*, December 2008.

[Nim10] Nimble Storage: CS Series models. http://www.nimblestorage.com/products/nimble-cs-series-family/. Data Sheet, 2010.

[NMRW02] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, Lecture Notes in Computer Science, 2002.

[NTD+09] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proc. of ACM Eurosys*, 2009.

[OAT+07] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and Philip C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. *IEEE MSST*, 2007.

[obj] ObjectStore Release 7.3 Documentation. http://documentation.progress.com/output/ostore/7.3.0/.

[OBS99] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[OOWZ93] Elizabeth J. O'neil, Patrick E. O'Neil, Gerhard Weikum, and Eth Zurich. The lru-k page replacement algorithm for database disk buffering. pages 297–306, 1993.

[Ora10] Oracle Corporation. Bringing Storage Efficiency to a New Level. An Oracle White Paper, September 2010.

[Owe10] Frank Owen. Storage Tiering vs Caching. Techvirtuoso.com, November 2010.

[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*, pages 68–78, 2004.

[PBA+05] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. *Proc. 20th ACM Symposium on Operating Systems Principles*, October 2005.

[PBK95] J. S. Plank, M. Beck, and G. Kingsley. Libckpt: transparent checkpointing under Unix. In *Proc. of the USENIX ATC*, January 1995.

[Pet09a] Mark Peters. Compellent harnessing ssds potential. ESG Storage Systems Brief, 2009.

[Pet09b] Mark Peters. Netapp's solid state hierarchy. ESG White Paper, 2009.

[Pet10] Mark Peters. 3par: Optimizing io service levels. ESG White Paper, 2010.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of ACM SIGMOD Conference on the Management of Data*, pages 109–116, 1988.

[Phi98] Barry Phillips. Have Storage Area Networks Come of Age? *Computer*, 31(7):10–12, July 1998.

[Pra02] Steven L. Pratt. EVMS: A Common Framework for Volume Management. *Proc. Ottawa Linux Symposium*, June 2002.

[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.

[QD02] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proc. of USENIX FAST*, 2002.

[RCP08] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. of USENIX ATC*, 2008.

[RO91] M. Rosenblum and J. Ousterhout. The Design And Implementation of a Log-Structured File System. In *Proc. of ACM SOSP*, 1991.

[Ros90] David S. H. Rosenthal. Evolving the Vnode Interface. *Proc. Summer USENIX Conference*, pages 107–118, 1990.

[RW93a] A L Reddy and J Wyllie. Disk Scheduling in a Multimedia I/O System. *Proc. ACM Conference on Multimedia*, pages 225–233, 1993.

[RW93b] Chris Ruemmler and John Wilkes. Unix disk access patterns. Technical report, HP Laboratories, 1993.

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

[SGM91] Carl Staelin and Hector Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, pages 45–52, 1991.

[Sil11] Carol Silwa. Sub-LUN tiering: Five key questions to consider. Search-storage.com Feature, February 2011.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Intl Workshop on Persistent Object Systems*, September 1992.

[SMK+93] M. Satyanarayanan, H. Mashburn, P. Kumar, David C. Steer, and J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of the ACM SOSP*, 1993.

[SPBW10] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proc. of USENIX FAST*, February 2010.

[spc] SPC specifications. `http://www.storageperformance.org/specs`.

[sql] SQLite. http://www.sqlite.org/.

[SS97] Margo Seltzer and Christopher Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. of the Workshop on HotOS*, 1997.

[SWS05] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration in Self-managing Storage Systems. In *Proc. of ICAC*, 2005.

[Tan10] Taneja Group Technology Analysts. The State of the Core Engineering the Enterprise Storage Infrastructure with the IBM DS8000. White Paper, 2010.

[Tec] Technical Review. Higher-Capacity Flash Memory. Online, http://www.technologyreview.com/.

[The07] The FreeBSD Documentation Project. *FreeBSD Handbook - Chapter 19: GEOM: Modular Disk Transformation Framework.* , 2007.

[The09] The PaX Team. PaX Address Space Layout Randomization (ASLR). *Available online at: http://pax.grsecurity.net/docs/aslr.txt*, 2009.

[tpl] tpl. http://tpl.sourceforge.net/.

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

[UAM01] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of IEEE MASCOTS*, 2001.

[UGB+08] Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon, and Raju Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. In *Proc. of IEEE HPCA*, February 2008.

[VD92] Francis Vaughan and Alan Dearle. Supporting large persistent stores using conventional hardware. In *In Proc. International Workshop on POS*, 1992.

[ver] Versant. http://www.versant.com/.

[VKUR10] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *Proc. of USENIX FAST*, 2010.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS*, 2011.

[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.

[WD92] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc of VLDB*, 1992.

[Wil92] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of ISMM*, 1992.

[Won80] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[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.

[WW02] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, 2002.

[Yue91] Minyi Yue. A simple proof of the inequality ffd(l) (11/9)opt(l) + 1, for all l, for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321331, 1991.

[Zad99] E. Zadok *et al.* Extending File Systems Using Stackable Templates. *Proc. USENIX Technical Conference*, June 1999.

[ZCD+10] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated Lookahead Data Migration in SSD-enabled Multi-tiered Storage Systems. In *IEEE MSST*, 2010.

[ZCL04] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions On Parallel And Distributed Systems*, 15(7):2004, 2004.

[ZCT+05] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberley Keeton, and John Wilkes. Hibernator: Helping Disk Array Sleep Through the Winter. *In proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[ZDD+04] Qingbo Zhu, Francis M. David, Christo F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *The 10th International Conference on High-Performance Computer Architecture (HPCA-10)*, pages 118–129, February 2004.

[ZLP08] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of USENIX FAST*, Feb 2008.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundarsn, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of ASPLOS*, October 2004.

Appendices

A EDT EVALUATION AND DISCUSSION

A.1 Evaluation

Our evaluation uses both a SPC-1-like [spc] benchmark workload and multiple production enterprise workloads from MSR [NDR08] to demonstrate that:

- In comparative evaluation, EDT-CA works to minimize cost, and EDT-DTM satisfies performance requirements while lowering power consumption.
- EDT's dynamic behavior and detailed resource consumption model help achieve its goal.
- Extent based dynamic optimization and consolidation are feasible in practice with little overhead.

A.1.1 Methodology

Comparison candidates.

We compare EDT to three alternate solutions:

1. *SAS* is chosen to represent current enterprise storage system deployments that predominantly use only high performance SAS drives. The configuration is derived

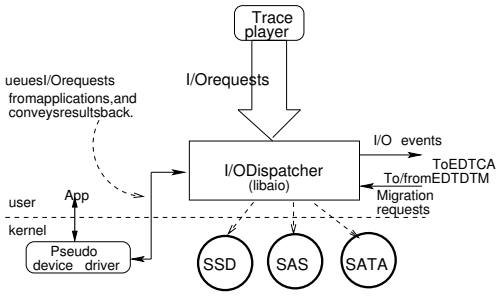


Figure A.1: Storage subsystem platform for evaluating EDT-CA and EDT-DTM.

using the capacity and **peak** performance (IOPS and bandwidth) requirements of the workload. Volumes are statically assigned to SAS arrays in a load-balanced manner.

2. *EST* (Extent-based Static Tiering) places extents on tiers statically to quantify the benefit from tiering. Configuration is performed as follows: at every epoch, the cost to place each extent on each tier is computed as done by EDT-CA using capacity, IOPS, and bandwidth requirements. An extent is then permanently placed on the tier that minimizes the sum of its instantaneous costs over all epochs. Once extents are binned into tiers, the number of devices for each tier is determined using that tier's peak resource consumption.

3. While SAS and EST illustrate the benefit from EDT's design choices incrementally (going from a homogeneous system to static tiering and then to dynamic tiering), we propose a third candidate to illustrate a different design decision in dynamic multi-tier systems—*IDT* (IOPS Dynamic Tiering) implements extent-based dynamic configuration and placement using a greedy IOPS-only criteria where higher IOPS extents move to higher IOPS tiers. This is in contrast to EDT that uses a combination of capacity, IOPS, and bandwidth in its placement algorithm.

Implementation.

Our test system is shown in Figure A.1. In addition to EDT, we implemented an I/O dispatcher that receives block I/O requests from applications, maps the logical block address to the physical device address, performs the corresponding I/Os, and communicates with the EDT components. Our *trace player* application issues block I/Os from a trace via a socket to the I/O dispatcher. To support real-world applications without modification, we implemented a pseudo block device

Device	Cost ($)	Power (Idle, Active)	Random IOPS	BW (MB/s)	Rtime (ms/IO)	Xtime (ms/KB)
SSD	430	0.5, 1	5000	90	0.2	0.01
SAS	325	12.4, 17.3	290	200	3.75	0.004
SATA	170	8.0, 11.6	135	105	9	0.009

Table A.1: Characteristics of devices used in the testbed.

interface. For the scope of this work, we use Linux's default *deadline* scheduler, and our measurement of context switch overhead when running through the pseudo device driver was negligible ($< 10\mu$s).

Experimental Testbed.

Our experimental platform consists of an IBM x3650 with 4 Intel Xeon cores and 4 GB memory acting as the I/O dispatcher. It is connected via internal and external SAS ports to 12 1 TB 7200 rpm 3.5" SATA drives, 12 450 GB 15K rpm 3.5" SAS drives, and 4 180 GB Intel X25-M SSD drives. Table A.1 shows the characteristics of these devices. The enclosures containing the drives are connected to a *Watts up? Pro* power meter. We report the disk power obtained by subtracting the baseline power used by the non-disk components of the enclosure (154 W).

Metrics.

To compare solutions, we evaluate static configuration results using capital cost and peak power consumption, and we evaluate dynamic behavior using the average and distribution of I/O latency along with dynamic power consumption. Peak power consumption is obtained using disk drive data sheets. Dynamic power consumption is measured using the power meter.

A.1.2 Parameter Selection

Extent size. Smaller extents use tier and migration-related resources more efficiently and enable faster response to workload changes, but also incur greater metadata overhead. Our approach was to pick the smallest extent size that incurs acceptable metadata overhead. Assuming metadata can have a reasonably small overhead of at most 0.0001% of the total storage capacity, and given 200 bytes/extent for metadata overhead (mostly from recording extent-level statistics) in our implementation, the smallest extent size our storage system can support is 20 MB. To introduce some slack we used 64 MB extents for our experiments.

Epoch duration. Shorter epochs allow quicker response to workload changes, but can also result in increased extent migration. As the epoch duration increases, the stability of extent characteristics increases due to averaging over longer periods and consequently the migration bandwidth overhead decreases. We picked epoch durations that resulted in migration bandwidth limited to a 10% fraction of the available array-pair bandwidth in the system[1]. This prevents migration from significantly degrading performance and ensures that migrations complete early within each epoch. For the MSR workloads this calculation resulted in a 30 minute epoch.

A.1.3 Synthetic Workload

This SPC1-like workload was chosen because it simulates an industry standard benchmark and provides a contrast to the MSR trace workloads. We ran the SPC1-like workload generator on a 1 TB volume at 100 BSUs for 30 min using an over-provisioned configuration (a 12 SAS RAID-0 array). We chose 30 min because the

[1]Medium to large scale tiered storage systems would typically perform simultaneous extent migrations across multiple array-pairs.

System	# of Disks	Energy	Cost	Avg RT
SAS	(0, 6, 0)	103.8 W	$1950	28 ms
EST	(2, 2, 1)	46.6 W	$1680	15 ms
IDT	(2, 1, 1)	29.3 W	$1355	21 ms
EDT	(2, 2, 1)	46.6 W	$1680	15 ms

Table A.2: Configuration for synthetic workload. The number of disks per tier is specified as (SSD, SAS, SATA). The average response time is obtained from running the configuration with 100 BSUs .

workload is quite static after a short startup period. The resulting trace was used to obtain the number of devices required per tier for different methods (Table A.2).

We observe that all the extent-based tiering configurations outperform SAS configurations in both capital cost and peak power consumption. EDT reduces cost by 14%, and peak power by 55% compared to SAS. Cost incurred to configure EST and EDT for this relatively static workload are similar. Although the IDT configuration seems to provide the least cost configuration, this is an artifact of rounding up required devices to the next higher integer. Using fractional devices, costs for EDT and IDT are much closer ($890 vs. $920). Note that in larger systems rounding effects will be less significant.

To confirm that EDT's lower cost is not at the expense of performance, we ran the SPC1-like workload for 30 minutes at 100 BSUs. Given the stability of the workload, migration overhead was minimal. We therefore chose an epoch of 5 minutes to complete the experiments quickly. The SAS scheme used 6 SAS RAID-0 array. Other schemes operated on individual disks. We started EDT and IDT with the entire volume in the SATA tier and allowed dynamic extent migration to reach optimal configurations over time. EST, which does not support extent migration, was started with extents in their most suitable locations as per the EST configuration.

Config	System	# of Disks	Energy	Cost
Equal Performance	*SAS*	(0, 16, 0)	276.8 W	$5200
	EST	(5, 2, 4)	82 W	$3480
	IDT	(4, 1, 4)	64.5 W	$2725
	EDT	(3, 2, 4)	81.6 W	$2620
Equal Cost	*SAS*	(0, 12, 0)	204 W	$3900
	EST	(4, 4, 4)	116 W	$3700
	IDT	(4, 4, 4)	116 W	$3700
	EDT	(4, 4, 4)	116 W	$3700

Table A.3: Configuration for *MSR-combined*. Configurations achieving equal performance depict improvement in cost and peak power. Configurations at equal cost are created for experimental ease. Number of disks in each tier specified as (SSD, SAS, SATA).

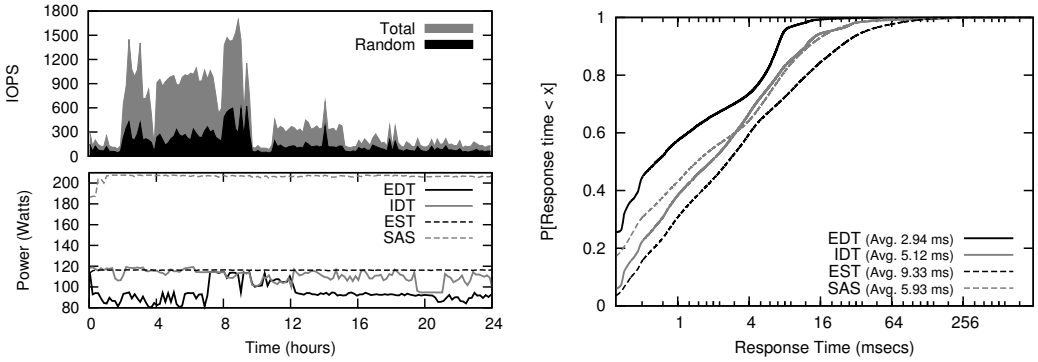


Figure A.2: I/O rate and power consumption (left) and response time distribution (right) for *MSR-combined*.

The last column of Table A.2 shows the average response times for 100 BSUs measured starting at the end of the first epoch, once the extent placements of the dynamic tiering configurations become effective. Given the workload's stability, results for EDT and EST are identical. They both achieve a 40% lower response time compared to SAS, and improve on IDT's IOPS only placement by 20%. Note that the dynamic power consumption in these experiments is similar to the peak power due to the lack of workload variation.

Production Workload Our next workload (*MSR-combined*) represents the more interesting class of real-world workloads, obtained by combining the I/Os to the 31

(out of 36) most active volumes of a production storage system [NDR08] for a total of 4580 GB. Including the remaining 5 volumes was not feasible given the hardware restrictions of our testbed.

Configuration outcomes. Configuration outcomes based on six days of the *MSR-combined* workload, shown as the "Equal Performance" group in Table A.3, indicate that the tiering configurations have lower cost compared to SAS. EDT incurs the lowest cost (50% reduction compared to SAS and 25% relative to EST). EDT's ability to effectively time share high-cost, high-performance tiers across extents and satisfy sequentially accessed ones with the SAS tier (instead of the SSD tier) results in more cost-effective configurations. Extents placed in the SATA tier (4336 GB) are mostly idle with random IOPS below 0.32, those in SAS (69 GB) are dominated by bandwidth higher than 1.45 MB/s and random IOPS less than 1.43, and the SSD extents (175 GB) have random IOPS between 1.45 and 858. Tiered configurations substantially reduce peak power when compared with SAS; IDTs greater use of the SSD tier (relative to SAS) makes it the most power-efficient.

Performance and Power outcomes. Not all of the equal performance configurations listed in Table A.3 were feasible on our experimental testbed due to hardware limitations. Consequently, we decided to switch to equal cost configurations (shown in Table A.3) to contrast performance at equal cost instead of cost at equal performance only for the *MSR-combined* workload. Later, we shall explore equal performance configurations for feasible subsets of volumes (Figure A.4). EDT's configuration was chosen as the base for all the tiering systems, and its configuration requirements were rounded up to integer number of arrays, each array consisting of 4 devices. SAS used only SAS drives for the same cost, split into 4 disk RAID 0 arrays. We then replayed day one from the seven day trace, the most active 24 hour

period of the *MSR-combined* workload. Both EDT and IDT were bootstrapped using a load balanced volume placement.

Figure A.2 summarizes the results of this experiment for the candidate solutions. First, we notice that the I/O response time distribution of EDT is clearly superior to the other three solutions, highlighting the importance of considering random IOPS, bandwidth, and capacity when making tiering choices. The average response time with EDT was 2.94 ms while those for the SAS, EST, and IDT were 5.12, 9.33, and 5.93 ms respectively. Further, the 95^{th} percentile response time for EDT was under 7.86 ms while the same for SAS, EST, and IDT were 19.31, 37.06, and 17.891 ms respectively. On average, EDT decreased the dynamic power consumption by 13% relative to its peak power, 55% relative to SAS and at least 10% relative to IDT and EST. This dynamic power savings result is likely to underestimate power savings observed in real deployments given that the workload was generated by consolidating multiple uncorrelated workload traces, which tended to reduce the workload variability that would enable dynamic power savings. Additionally, the experiment was done over the most active period, which required most devices to be active for performance. Further, all the configurations here are sized to meet the observed workload. Typically, however, storage purchases are made to accommodate future growth and hence over-provisioned to begin with, resulting in more dynamic power savings.

Analysis. We illustrate how EDT achieves its superior performance using two example extents chosen from the experiment and contrasting them with IDT. Figure A.3 shows the sequential and random IOPS over time for two extents along with the tier they are placed in. For extent A (top graph), both IDT and EDT move the extent from the SATA tier (the default initial location) to higher performing tiers when the total IOPS requirements increase. However, IDT allocates the SSD

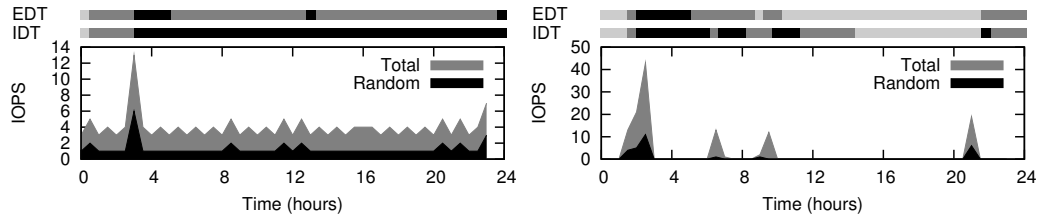


Figure A.3: Contrasting extent migrations for EDT and IDT. The two upper lines denote extent placement for the different algorithms. Black is SSD tier, dark grey SAS and light grey SATA.

tier starting from hour 3 on account of the exponentially weighted moving average (EWMA) of total IOPS whereas EDT allocates the SSD tier only when the EWMA of *random* IOPS of the extent is high. Thus, EDT can better capitalize on the superior sequential performance of the SAS tier to minimize capital costs during configuration and sustain performance during operation. Extent B (bottom graph) illustrates similar behavior during predominantly sequential accesses. Further, both EDT and IDT rightly move extent B into the SATA tier when it becomes idle, aiding in power savings. Thus, EDT is successfully able to pick the best tier for an extent's workload and relocate it when the requirements change. Regarding the overheads for this migrations, both EDT and IDT migrated around 120 extents per epoch, using an average bandwidth of 42 MB/s which only represents 3% of the total available.

Workload	**Volumes**	**Cap (GB)**	**Accessed**
server	hm, mds, prn, prxy, stg, ts, wdev, web	1650	30%
data	proj, rsch, usr	3719	34%
srccntl	src1, src2	904	29%

Table A.4: Sub-workloads derived from MSR.

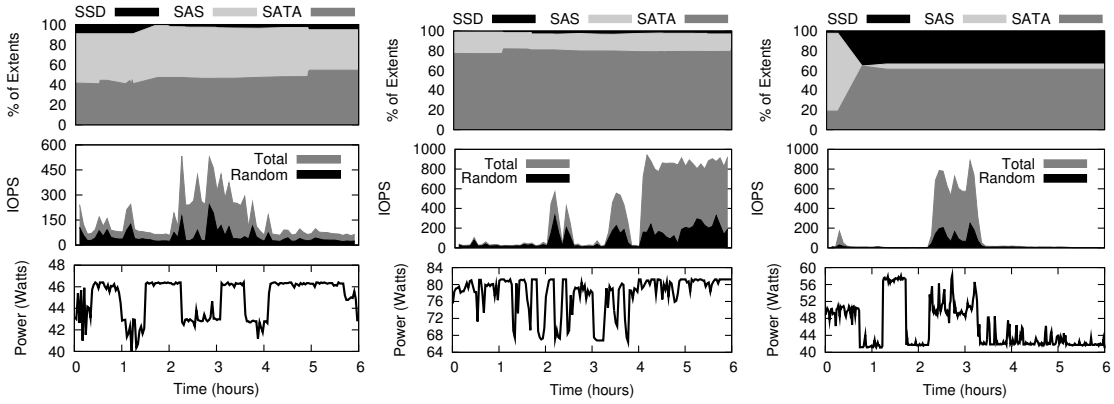
Varying the workload. To analyze the sensitivity of the various algorithms to workload characteristics, we grouped volumes from the MSR workload as specified in Table A.4 to create the *server*, *data* and *srccntl (source code control)* workloads.

Workload	System	# of Disks	Energy	Cost
server	*SAS*	(0, 6, 0)	103.8 W	$1950
	EST	(2, 1, 2)	42.5 W	$1525
	IDT	(2, 1, 1)	30.9 W	$1355
	EDT	(1, 2, 1)	47.2 W	$1250
data	*SAS*	(0, 10, 0)	173 W	$3250
	EST	(2, 2, 3)	71.4 W	$2020
	IDT	(1, 2, 4)	82 W	$1760
	EDT	(1, 2, 4)	82 W	$1760
srccntl	*SAS*	(0, 6, 0)	103.8 W	$1950
	EST	(2, 3, 1)	65.5 W	$2005
	IDT	(2, 2, 2)	59.8 W	$1850
	EDT	(2, 2, 2)	59.8 W	$1850

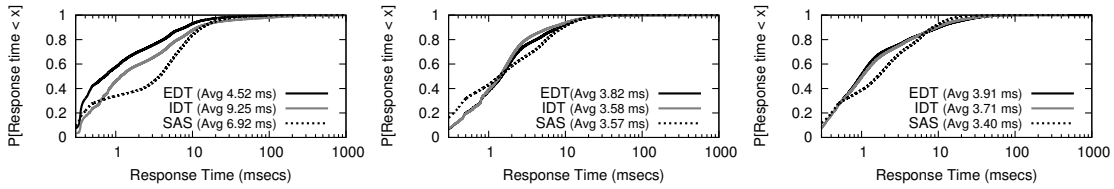
Table A.5: Configuration for MSR sub-workloads. Number of disks in each tier specified as (SSD, SAS, SATA).

Configuration outcomes for each sub-workload using SAS, IDT, and EDT are presented in Table A.5. As with *MSR-combined*, the dynamic tiering solutions are able to configure both lower-cost and lower-energy systems when compared with SAS and EST. Further, in the case of the *server* workload, EDT optimizes the configured system cost with a single SSD relative to the two SSDs recommended using IDT. Given that EST had significantly inferior performance for *MSR-combined*, we did not consider it for further analysis.

Figure A.4 shows EDT's dynamic power consumption and extent distribution across tiers over time, as well as its response time distribution relative to IDT and SAS. First, unlike *MSR-combined*, these workloads do have substantial periods of lower utilization. Consequently, in addition to improving the capital cost and peak power consumption, EDT's dynamic consolidation allows dynamic power savings of as much as 15-31% relative to its peak power across the three workloads. The extent distribution is quite different across the workloads. EDT uses the SSD tier substantially for the *srccntl* workload. IOPS-wise one would think that the workload should be completely consolidated to the SATA; however, EDT leverages the fact that the



(a) EDT's extent distribution, I/O rate, and EDT's power consumption over time.

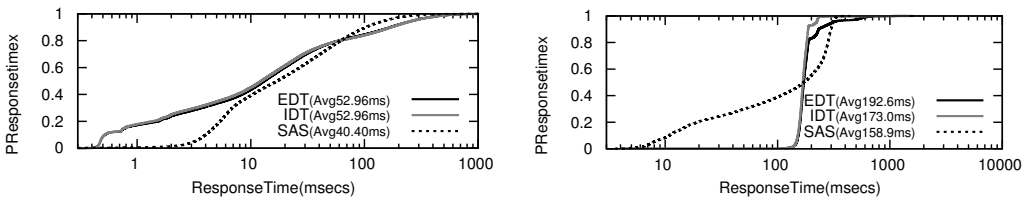


(b) CDF of response times

Figure A.4: Replaying 6 hours of the MSR sub-workloads. First column is *server*, second *data*, and third *srccntl*.

SSD tier offers improved energy efficiency for up to 40% of the extents. The SAS tier was most used for *server*, in particular between hours 2-4 when sequential activity dominates. The *data* workload predominantly utilizes the SATA tier (as evidenced in the configuration outcome) since the IOPS per extent for most extents is very low, easily accommodated using SATA devices. Finally, in this equal performance configuration experiment, the response time performance with EDT is either similar or better than the SAS and IDT schemes across the workloads.

Adversarial Workloads Finally, we measure the impact of using EDT with workloads completely different than the one it is provisioned for. We used the configuration obtained for the *srccntl* workload (in Table A.5), and instead of the trace from that workload, we ran two separate synthetic workloads for two hours each: (1) a uniformly random workload at 400 IOPS, where each I/O is issued to a



(a) Uniformly Random Load (b) Varying Hot Extent Set

Figure A.5: Extent distribution and CDF for the adversarial workload.

random page in the system. (2) a workload at 500 IOPS, where I/Os are issued to a chosen set of 10 hot extents initially in the SATA tier and this set changes every minute.

Figure A.5 depicts the distribution of response times for both workloads. The uniformly random workload yields a 31% higher average response time for EDT and IDT compared to SAS. This can be attributed to the constant migration I/O moving extents away from the throttled SATA tier to both SAS and SSD tiers. Interestingly, we see only a 21% penalty for EDT in the second workload. Analysis shows that throttling of the newly active extents was promptly detected and the extents were migrated quickly to the SSD before they became cold. As illustrated by these examples, EDT can handle unexpected workloads using its throttling detection/correction techniques without major performance penalties.

A.2 Discussion

Extending the resource consumption model In this work we assumed RAID-0 arrays when estimating how much resource on a tier is consumed by a given workload. In commercial applications of EDT, more sophisticated models will be needed to estimate resource consumption in arrays with different RAID levels. Such models do already exist in the industry, so we believe incorporating this capability will be straightforward. Also, for the scope of this work, we assume that all arrays are at

the same reliability level, and hence migrating data across arrays is not restricted. However, it is feasible to remove this constraint by observing policies to limit the migration targets of extents. Finally, the resource model may need be enhanced to better model the behavior of disks servicing multiple sequential IO streams in parallel. The current model does not account for degradation in sequential performance that may occur when a disk needs to service multiple sequential streams at once.

Disk power fraction in the overall energy of a storage system. The chief dynamic energy-saving technique proposed in this work is powering down empty disk drives. However, we find that in today's commercial storage systems, disk drives typically consume ∼50% of the total storage system energy [IBM10] while the rest is consumed by other components which do not currently have the capability of varying their energy consumption according to workload. As these components overcome this limitation, our energy-saving techniques can be extended to include them, leading to a more energy proportional system and lower overall operating costs.

Applicability. The target domain for EDT is primary storage systems where response time is critical. Archival applications where response time is not as critical may be better served with existing solutions using policy-based migration and power-saving storage such as spun-down disk or tape. Also, EDT will be most effective when the working set and I/O intensity are somewhat stable with some variation. When the workload is static, dynamic migration will not take place but consolidation will still be beneficial if the system is not capacity bound.

VITA

JORGE GUERRA

2005	B.S. Computer Science
	Universidad Simón Bolívar
	Sartenejas, Venezuela
2007-2012	Doctoral Candidate
	Florida International University
	Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei (2012). *Software Persistent Memory.* Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12).

Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini and Raju Rangaswami. (2011). *Cost Effective Storage using Extent Based Dynamic Tiering.* Proceedings of 9^{th} USENIX Conference on File and Storage Technologies (FAST '11).

Jorge Guerra, Wendy Belluomini, Joseph Glider, Karan Gupta, Himabindu Pucha (2009). *Energy Proportionality for Storage: Impact and Feasibility.* Proceedings of SOSP Workshop on Hot Topics in Storage and File Systems (HotStorage '09).

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 7^{th} USENIX Conference on File and Storage Technologies (FAST '09).

Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcón, 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), held in conjunction with IEEE HPCA.