The Pennsylvania State University
The Graduate School
College of Engineering

A PROTEAN ATTACK ON THE COMPUTE-STORAGE GAP IN HIGH-PERFORMANCE COMPUTING

A Dissertation in
Department of Computer Science and Engineering
by
Ellis H. Wilson III

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2014

The dissertation of Ellis H. Wilson III was reviewed and approved* by the following:

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Thesis Advisor, Chair of Committee

Padma Raghavan
Professor of Computer Science and Engineering

Wang-Chien Lee
Professor of Computer Science and Engineering

Christopher Duffy
Professor of Civil Engineering

Raj Acharya
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Distributed computing, in particular supercomputers, have facilitated a significant acceleration in scientific progress over the last three-quarters of a century by enabling scientists to ask questions that previously held intractable answers. Looking at historical data over the last 20 years for the top supercomputers in the world, we note that they have demonstrated an amazing doubling in performance every 13.5 months, well in excess of Moore's law. Moreover, as these machines grow in computational power, the magnetic hard disk drives (HDDs) they rely upon to store data to and retrieve data from double in capacity roughly every 18 months. These facts considered in concert provide a foundation for the recent data-driven revolution in the way both scientists and businesses extract useful knowledge from their increasing datasets.

However, while computation and capacity potential for these machines is growing at a breathless rate, a disturbing but oft-ignored reality is that the ability to access the data on a given HDDs is shrinking by comparison, doubling only once every decade. In short, this means although the capability to process and store the data scientists and businesses are so excited about is here, the ability to access that data (a prerequisite for processing it) falls behind year in and year out.

Therefore, the focus in this thesis is to find ways to limit or close the annually widening compute-to-bandwidth gap, specifically for systems at scale such as supercomputers and the cloud. Recognizing that this problem requires improvement at numerous levels in the storage stack, we take a protean approach to seeking and implementing solutions. Specifically, we attack this problem by researching ways to 1) consolidate our storage devices to maximize aggregate bandwidth while enabling best-of-breed analytic approaches, 2) determine optimal data-reduction techniques such as deduplication and compression in the face of a sea of data and a lack of existing analysis tools, and 3) designing novel algorithms to overcome longevity shortcomings in state-of-the-art alternatives to magnetic storage such as flash-based solid-state disks (SSDs).

Table of Contents

List of Figures

List of Tables

Acknowledgments

I proceed with great caution in writing the acknowledgments for my Ph.D. below. On the one hand, it would be utterly callous to not pen down some form of thanks to the many people who have contributed in one way or another towards the successful completion of this milestone in my life. However, on the other, and perhaps what I fear more, it seems almost impossible to enumerate and thank each and every influence that has brought me to this moment in time. Therefore, to whomever I errantly exclude, please forgive the flighty memory of an academic.

This dissertation and my Ph.D. on the whole are in no small part attributable to the interactions with and inspiration from three distinct groups in my life: fellow academics, key figures in my distance running career, and my friends and family, which I address in order below. Please note that persons addressed within categories generally follow first chronological impact–no assignment of greater or lesser import should be derived from their ordering.

While my love for computing in general stretches as far back as my very early teens, my interest in high-performance distributed computing was born during my undergraduate sophomore year when I read "Engineering a Beowulf-Style Compute Cluster," a free online book by Robert G. Brown (rgb). Although I cannot recall how I first stumbled upon it, my thanks must be given to rgb for writing (and freely providing) the text and those on the Beowulf mailing list for their helpful support and advice as my curiosity for distributed computing first blossomed.

This curiosity first struggled to find appropriate soil (i.e., free machines for me to build a cluster with) until I was introduced to Michael Prushan, a professor of chemistry at my undergraduate institution, La Salle University. During my junior and senior years as I learned by trial-and-error how to build a cluster, he was an unparalleled advisor of my research with a lust for problem-solving and teaching I will never forget. In working with him I was offered my first exposure to interdisciplinary research, an experience every computer scientist must have to fully appreciate and understand the scope of his own field. Equally critically, he introduced me to Robert Levis, another chemistry professor and director of Temple

University's Center for Advanced Photonics Research (CAPR). It was during my undergraduate internships with Robert at CAPR that I managed my first sizable cluster, learned (again, mostly by trial-and-error) to write applications that would scale, was exposed to the basics of machine learning, and came to grips with the raw difficulty of problems academics face (and fall in love with) on a day-to-day basis. This problem difficulty and corresponding progress by tiny, deliberate steps was addictive, and a key instigator in my application to Ph.D. programs. My sincere thanks to you both for your advisement and education in what I consider my first years as a scientist.

To this day it is not entirely clear to me why my doctoral advisor, Mahmut Kandemir, called me with an acceptance to a top-thirty computer science Ph.D. program, but I am eternally indebted to him for giving me a chance. I will admit I was not the classic graduate student in any way, shape, or form, having concurrently started and helped to run a company throughout my five years in the program, getting married after my first year, and living and researching remotely for nearly half of my program's duration. However, through all of my struggles and what proved to be the hardest (and most rewarding) intellectual adventure I have had to date, he was, and still is, a compassionate advisor who knows how to push his students to their finest while respecting their limits and their private life. I imagine it was in this wisdom that, when I was floundering the most transitioning from classwork to solely research midway through the program, he introduced me to a fellow graduate student by the name of Myoungsoo Jung (MJ). I am reasonably certain that without the company of and collaboration with MJ, my Ph.D. may not have continued to fruition, or in the best case, this dissertation would be a shell of what it is now. MJ not only served as a collaborator and friend during the second-half of my Ph.D., but also as an advisor in many ways as I refined my research and publication process, and as a hallmark example of what a great scientist looked like up close. To both Mahmut and MJ, unspeakable thanks for your patience, support, and encouragement as I grew into the scientist I am now.

Mahmut's wisdom and respect for private life as an advisor also led to my introduction to Garth Gibson. Garth provided some of the most challenging and interesting internships during the last three summers of my Ph.D., and a good chunk of the research presented in this dissertation is a direct result of his advisement. Working with Garth refined my love for distributed computing into a love specifically for the equally distributed storage that backs it, and researching and working under the guidance of the "father of RAID" was an unparalleled opportunity–one which I will appreciate for years to come in my tenure at Panasas. Just as it was with Mahmut, my thanks for giving me a chance summer after summer and now full-time, Garth.

Moving from the academic influences that must be acknowledged for their

more obvious impact on this dissertation, less direct but in many ways equally potent influences from my running career must be noted and thanked for their development of my personality that was key in getting through the Ph.D.. In both distance-running and a Ph.D., there is a requirement to be able to perform under immense fatigue and to cope with short-term failure given a vision of long-term improvement and ultimate success. Beginning in high school, David Lapp and Keith Andrews served as my cross-country and track-and-field coaches and developed me from a very unathletic and untempered youth to a conference champion and a competitor at the state-level. Under their care I was not only personally tempered both mentally and physically, but was given my first chance at leadership as captain of the team my senior year. And so, to coaches Lapp and Andrews, my sincere appreciation–I would have been spent in the first mile of this Ph.D. without your impact on my personality.

Beyond high school, because of my unwavering devotion to distance running and desire for a full career in the sport, I met the only character in this list that I am forced to acknowledge for the personal development I underwent as a result of all that was wrong with him and his coaching of myself. Marcus O'Sullivan, my coach for a very short time, embodied all that is counter to that of a scientist. During even that brief time under his guidance he evidenced a deep hatred for critical thought, discouraged question, demonstrated an uncanny willingness to go along with even senseless orders from those above simply due to systematic problems, and ultimately exuded a toxic elitism that could not be more in contrast with what a great coach and leader should be. In any moment of difficulty during this Ph.D. or any run I have been on since I left that team I have drawn strength from the memory of all that he was and all that I refuse to become. I am forced to thank you for that, and that alone, Marcus.

Last, to the man who welcomed me with open arms after my tenure under Marcus, embodied all that I knew a good coach could be, and developed me into the best runner I ever was and likely ever will be, thank you, Charles Torpey. Even on our first meeting it was plain I was at home on your team, and although you sought for our team to be the best, the social sludge of individual elitism was nowhere to be found. You taught through sarcasm, suffered with us in our losses, rejoiced with us in our successes, and led us with an iron will. My only regret is that I did not spend my entire undergraduate running career under your guidance and vision as a coach, and that you are no longer living to provide further guidance as I close this chapter of my life. I have no doubt that you would be proud of me for my accomplishments here, different as they may be from distance running, and my unending thanks go to you for picking me up and piecing me back together at a time in my life when I felt incredibly alone and fragile.

Completing my acknowledgments, I must note the huge impact my friends and

family have had on my person as a whole and my development into a scientist throughout this Ph.D.. To all of my longtime friends and the new additions to our wonderful group, knowing I will see and enjoy your company no matter how far we get from each other has been an immense motivator during this Ph.D.. In particular, for your encouragement, the many discussions I have had with you regarding this academic adventure, and your faithful friendship during this time, I wish to thank Bill Dowd, Maria Allegretto, Matt Gallis, Emily Smith, Alex Middleton, Ryan Miller, Sue Speck, Jon Jinks, Emily Myers, and Mike Motily. You all have been there through thick and thin for me, and I am indebted more than a few beers for it.

To my family and particularly my parents, Ellis and Barbara Wilson, whom I frequently spar with, infrequently understand, and rarely appreciate as much as I should–a big thanks. As much as it pains me to say so, I have inherited many wonderful traits from you, both scientific and creative, and they have served me extremely well over the past five years. To my inherited family, especially Ed and Jean Allegretto, my equal thanks. You have treated me as your own son in some of my most sincere times of need, and without any expectation of repayment. And, only partially in jest, a big thanks to the unspoken (in all forms) collaborators on all of my papers: Henry the harrassing turtle, Philip the egregiously large goldfish, and Bear the fur-issuing Husky-Shepherd mix. Last, to my wife who has been with me for over a decade and has helped me over every pebble and boulder of this journey, my love and my thanks, Maria Allegretto. I would never have seen the starting line much less the finish to this Ph.D. without your help, advice, and encouragement. You have been a driving force in my life, a catalyst of change that has kept me on my toes, have pushed me to the brink of my capability, and caught me when I began to fall. My thanks would cover the rest of these pages if I were forced to quantify it, and so, I instead:

Dedication

For, and in large part because of, the two strongest women I have had the pleasure to cross paths with in this life:

Maria Allegretto, my best friend, wife, bedrock of my sanity, fellow academic, and perpetual reminder of what really matters.

Ann (Melnick) Wilson, my grandmother, feverish advocate of education, unbreakable feminist, and known troublemaker now relegated to memory.

Gopher Guts

You were sitting at the gate awaiting spirits and provisions,
I was privy to a headache over pirouetting innards,
In the mirror sweating pitchers, who's there simian or lizard?
As it were there is a disappearing difference,
In ambition and material;
Antiquated gentleman outlaws reduced to a ferris wheel of vitriol.
Move as a godless heathen;
        black gums, tooth gone, bootleg Yukon Cornelius
I'm a - ksshht! That's better, here we here we go,
Disenchanted face printed on a zero-dollar bill,
Got a little plot of land where authority isn't recognized,
Contraband keeping the core of his Hyde Jekyll-ized,
Check! Never mind a misanthrope vying for affection
        to the wretched sound of mysticism dying,
It is something he must handle on his own;
        the wind blown way, wanna win? Don't play.

Excerpt from Gopher Guts on Skelethon – Aesop Rock

Chapter 1
Introduction

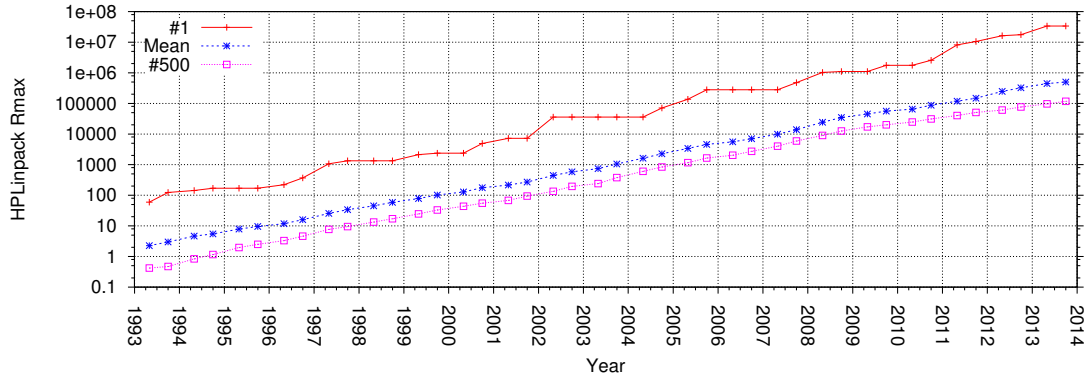
In 1987 in his book "Empirical Model-Building and Response Surfaces" George E.P. Box famously stated, "Essentially, all models are wrong, but some are useful." [1] This quote was supposedly adapted by Peter Norvig in the Wired article, "The End of Theory: The Data Deluge Makes the Scientific Method Obsolete," to read instead: "All models are wrong, and increasingly you can succeed without them." [2] Fortunately, Norvig didn't make this flawed statement, as he corrects on his site [3]:

> "[I]f the model is going to be wrong anyway, why not see if you can get the computer to quickly learn a model from the data, rather than have a human laboriously derive a model from a lot of thought."

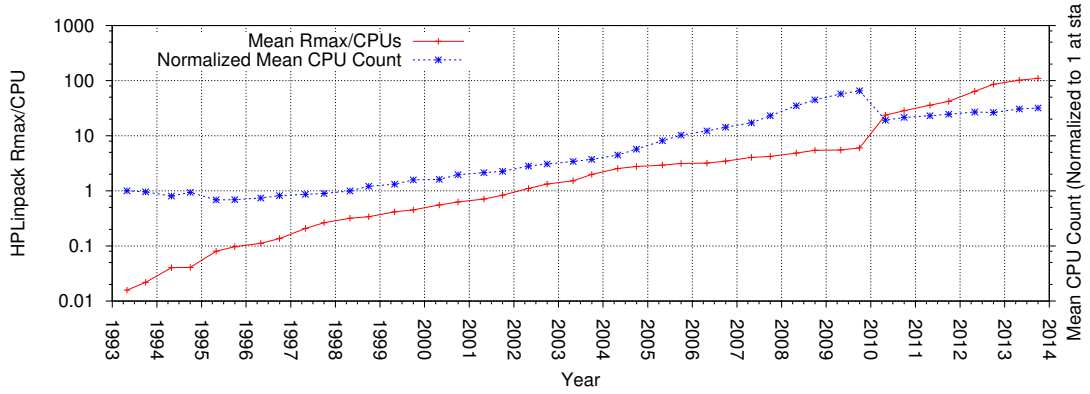
This mentality represents the slow but steady change that has occurred in how we, as brilliant but comparitively sluggish human beings, pursue problem-solving with a machine (or a supercomputer) by our side that doubles in speed and storage capacity every roughly one and a half years. Each of those epochs it becomes twice as efficient to spend our time "teaching the computers to learn" rather than trying to solve the problems directly ourselves.

1.1 Problem Statement

Unfortunately, one critical consideration is regularly downplayed while the focus remains on speed and capacity: Storage Bandwidth. Storage bandwidth is the very ability to get those widening seas of valuable data to and from the processor – limit it and the capability of the machine to extract knowledge from data is immensely handicapped.



(a) TOP500 Rmax



(b) TOP500 Rmax by CPU Growth

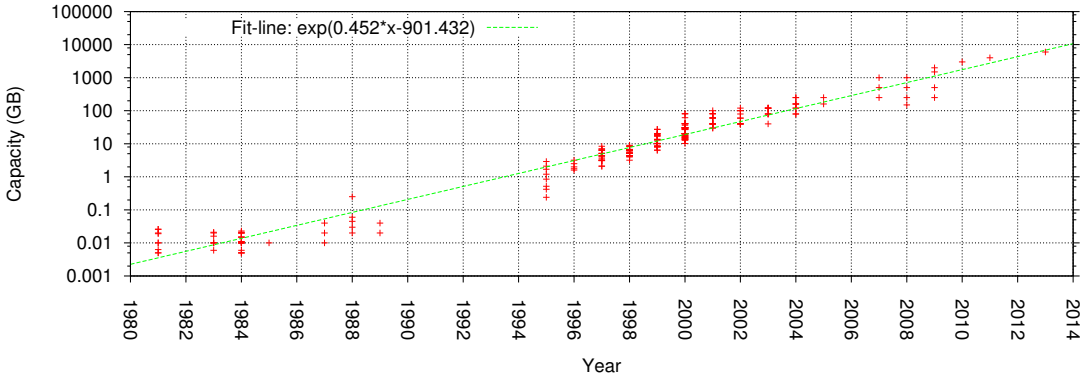
Figure 1.1: Biannual performance on the defacto High-Performane Linpack (HPLinpack) benchmark used to rank the top 500 supercomputers in the world. (a) shows raw megaflops for systems from 1993 to present for the first-ranked, last-ranked, and the mean of all 500 machines, demonstrating a super-Moore's-law doubling rate every 13.5 months. (b) normalizes the mean flops result over the same time-span against the mean number of CPUs in the systems for the red line, and shows raw CPU growth normalized to 1 for the first data collection in 1993 on the blue line. This graph explains the super-Moore's-law doubling rate, as the red line doubles once every 22 months, very near to Moore's-law, and raw CPU count doubles once every three years. Combined they account for the higher doubling rate in (a).

To appreciate the gravity of this situation, let us examine the most powerful (known) computing resources in the world: The supercomputers on the TOP500 list. [4] As demonstrated in Figure 1.1a, members of the elite TOP500 list have shown consistent and appreciable compute growth over the many years since its incarnation, *roughly doubling every 13.5 months*. This graph lays out the performance on the canonical HPC benchmark, High-Performance Linpack (HPL) [5], for the #1

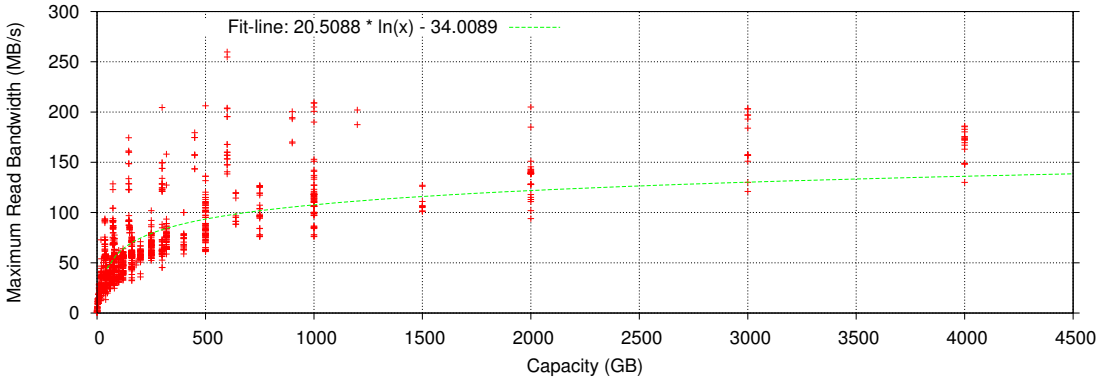
supercomputer, #500 supercomputer, and an average performance over all 500. Interestingly, a doubling rate of once every 13.5 months well exceeds Moore's law, which states transistors should double once every two years [6], during which time performance roughly follows that increase.

Expecting this disparity to result from concurrent increases in raw processor counts being added to the supercomputers in use over time, we plot normalized mean CPU count (no accelerators are considered) as the blue line, where 1 is the first mean taken in 1993. Note that in the 1993 to 1997 era, CPU counts reduced slightly as vector processors in big iron dominated, and from around 2005 to 2010 the processor counts were overstated as the TOP500 list reported core count as equivalent to processor count. This was corrected in 2010, where we see the slope of the processor count growth return to a pre-2005 angle. Overall, we see that every three years twice as many processors are used in the top 500 supercomputers across the globe, which accounts for the super-Moore's-law effect observed on the overall performance statistics in the Figure 1.1a. Correspondingly, when we divide the mean performances shown in that graph by the mean CPU counts, we find that the adjusted doubling rate (as shown in the red line) returns to a sane once every 22-months, very near to Moore's law.

Moreover, magnetic hard disk drive (HDD) capacity, the space required to store more and more valuable data going into and coming out of these rapidly improving machines *doubles once every 18 months* as shown in Figure 1.2a, which plots HDD capacity growth over the last 30 years for over 200 HDDs. This is promising as it out-paces Moore's law and nearly meets the 13.5 month doubling rate of the TOP500 list, however, just having the improved ability to store the data is not enough – one must also be able to get at that data (or put data to that storage) at scalable speeds, which brings us to bandwidth. Plotting the capacities and bandwidths for sequential read benchmarks for *over 1500 HDDs* from the early 1990s to 2013 (year first released is extremely difficult to attain for this data, and the data in the first graph did not report bandwidth), we can see that bandwidth scaling has entered a phase of linear growth, if not outright stagnation. Fitting those over 1500 drives to a curve and correlating it to the first graph where we ascertained that capacity doubles every 18 months, we arrive at a very sobering conclusion about hard disk bandwidth improvements: *Magnetic HDD bandwidth should only, in the best case, be expected to double once per decade.* This fact needs



(a) HDD Capacity Growth



(b) HDD Bandwidth Growth

Figure 1.2: Capacity and bandwidth growth over the years. (a) shows capacity growth from records of over 200 hard drives from 1980 to present, demonstrating a doubling effect every 1.5 years. (b) plots maximum read bandwidth results for over 1500 hard drives ranging over the same time-span, but correlates bandwidth to capacity as year first released was not available for the results. Correlating (a) and (b), we see that bandwidth enjoys a far lower improvement rate, only doubling once every decade, and appears to be reaching a stagnation point.

to be considered aside the reality that capacity is reaching super-paramagnetic limits [7], and the technologies that are currently being put forth to cope with these such as Heat-Assisted Magnetic Recording (HAMR) [8], Bit-Patterned Magnetic Recording (BPMR) [9], and Shingled Magnetic Recording (SMR) [10], are all expected to retain present-day bandwidths in the best case, and reduce them in the worst.

Put succinctly, although the compute power of HPC systems continues to grow at exceptional rates, and the disks used in these systems provide necessary capacity gains to meet computation increases, the very ability to get at the data on the disk

is proportionally shrinking year by year. *A decade ago, it took a mere 30 minutes to read all of the data off of a HDD – today it takes over 11 hours for the largest commercially-available disks – in three years it is perfectly reasonable to expect it to take a full day to read all of the data off of a single HDD.* Therefore, in order to keep up with rapid computation and capacity gains and avoid this bleak future, novel ways of employing the storage, manipulating the data that is stored within, and making improvements to the very storage devices themselves must arise to cope with this immense (and ever increasing) gap.

1.2 Thesis Statement

Exploring solutions to this compute-storage gap in the large-scale domain (i.e., HPC and Big Data) is the focus of this dissertation, and we take a multi-pronged approach to researching and engineering around it. These prongs are spread between the three obvious ways to cope with the reality of decreasing relative bandwidth in the near future:

- Reduce device fragmentation – assure all of your storage devices are working in concert
- Reduce the capacity occupied by the data being moved
- Attempt to use fundamentally different storage devices than traditional HDDs

By first reducing device fragmentation we achieve performance and capacity gains for the aggregated file system that are magnified by capacity reduction and a move to fundamentally different storage devices. However, this ordering is not necessarily required: Each of these approaches has a magnifying impact on the remainder, no matter which you choose to implement first. Nevertheless, the more of these three administrators and researchers consider in tandem to cope with bandwidth losses, the larger the benefits will be. Therefore, put more specifically, our tactics can be categorized into *data consolidation, data reduction, and storage device improvement*, and this dissertation follows this organization, respectively being chapters 2, 3 and 4.

1.2.1 Data Consolidation

In the first prong of this research we consider the fracturing landscape of modern storage. As various frameworks to cope with "Big Data" arise to provide simpler and more scalable ways of analyzing large pools of data, they also espouse (often require) divergent hardware configurations. This results in increasing amounts of the aggregate storage at an organization breaking into distinct pools. For instance, in web-scale companies real-time data tends to be kept in low-latency, enterprise storage solutions such as Network Attached Storage (NAS) arrays and managed by Relational Database Management Systems (RBDMS), while ad-hoc analytics is performed on a separate, mirrored pool of this data distributed across commodity machines and managed by a Big Data framework like Hadoop. So, not only is the overall bandwidth available to the organization split into two between the distinct systems, but copies need to be frequently performed to facilitate accurate and updated analytics, incurring unnecessary load on the systems in terms of CPU, network, and storage resources.

Therefore, in this work we examine an increasingly popular configuration: enterprise NAS storage aside an Apache Hadoop instance. Apache Hadoop, arguably the most popular "Big Data" analytics framework, enables programmers to leverage the rapid time-to-result programming framework MapReduce over distributed, commodity storage co-located with the compute. However, this comes with all of the downfalls of distinct systems as previously mentioned, so we examine how one might integrate Hadoop MapReduce into and on top of a NAS solution to enable consolidated storage and data. We demonstrate three different architectures enable converged storage, and explore performance and reliability impacts. Finally, finding all three to fall short in some regard or another, we design, implement, and propose a Reliable Array of Independent NAS File System (RainFS) to achieve the best of all architectures.

1.2.2 Data Reduction

While getting the data all into a single storage system to maximize aggregate bandwidth available and eliminate the need for capacity and bandwidth consuming copies is a critical first step, in our second research prong we take a different tact to solving this problem: data reduction. As demonstrated in Figures 1.1 and 1.2,

increasingly systems will have "left-over" compute cycles while they wait for storage to keep up. Therefore, we expect data reduction techniques like data compression and deduplication, while computationally expensive (in certain cases) today, may very well be cheap relative to waiting on storage in future systems. Numerous compression techniques have existed for years, and deduplication is a fairly well established technique at this juncture as well.

However, compression only works for certain types of data, and deduplication similarly will only provide benefits if real duplication exists. Employing these techniques on a pool of data that does not benefit from them will in the best case just expend needless compute, and in the worst exacerbate the already strained storage bandwith. Therefore, it is critical to enable the system administrator to evaluate her data on various compression and deduplication algorithms (and their many tunables) to ascertain if those techniques are effective for her data and if so, what parameters provide the biggest benefits. Ironically, there do not appear to be any easy tools for system administrators to evaluate whether deduplication and/or compression will work for their data short of turning these techniques on for the entire storage system, which may seriously disrupt production use of it, or acquiring a demo storage system from a vendor, which is an awkward and expensive experiment.

To fill this gap, we design a two-stage tool called TreeChunks that is built to scale and evaluates the efficacy of various compression and deduplication algorithms on a specified data pool. In the first stage high-performance C code is employed to rapidly chunk through the data and execute compressors and build a deduplication fingerprint database. In the second stage perl code is used to aggregate statistics from the distributed first-stage runs on compressors, and to count fingerprints in the deduplication database. Further, the second stage also builds information-rich diagrams, which depict with varying colors and node sizes just what portions of the file system benefit the most from the many compression and deduplication algorithms and the permutations of their parameters. This enables the system administrator to evaluate these techniques on their data inexpensively and to translate these statistics (e.g., gzip provides 50% savings) into value-rich organizational findings (e.g., the engineering teams data enjoys 75% savings from gzip while the rest of the organization only enjoys 10%).

1.2.3 Storage Device Improvement

Having consolidated the data into a single high-performance storage system and enabled ways to evaluate if state-of-the-art data reduction techniques effectively can boost bandwidth by shrinking the data itself, we now shift our focus to the lowest-level of the storage system: the storage device itself. The most popular storage device alternative to traditional magnetic spinning HDDs, NAND-flash-based solid state disks (SSDs), enable orders of magnitude lower latencies, lower power, and much higher bandwidths than their spinning brethren.

However, these devices suffer from two main issues: density and wear-out. In the first case, while it has been improving dramatically in recent years, they traditionally lack the raw capacity HDDs enjoy. In the second, these devices employ NAND flash blocks that must be erased before they are written to again, and these erases can only be performed a limited number of times before the block can no longer reliably retain data. Furthermore, as manufacturers push for higher densities to meet consumer demand and expectations, the wear-out issues have come to the fore again as modern cell technologies such as triple-level cell use smaller feature sizes and enjoy as much as two orders of magnitude lower lifetime than their larger feature-sized grandfathers, single-level cell, did.

Nevertheless, SSDs yet remain a powerful ally towards reducing the compute-storage bandwidth gap, so in this research we explore ways to enhance longevity for modern, low lifetime SSDs. Specifically, we bring forward a technique we dub "ZombieNAND," which leverages the reality that single-level, multi-level, and triple-level cell technologies are all just logical differentiations. Therefore, we see if it is possible to transition a dead block down one bit-level and regain lifetime. Finding that it is, we design a physically-accurate simulator to explore the magnitude of the lifetime gains available and if any performance implications exist. Simulating these drives from pristine state to death, we demonstrate over ten-fold improvements in lifetime and in some cases reductions in latency in excess of 50% over a wide variety of synthetic benchmarks and real trace-driven workloads.

Chapter 2
Data Consolidation: Enabling Big Data Computation atop Traditional HPC NAS Storage

2.1 Introduction

Cluster computing specialized for processing massive virtual and physical sensor data, one definition of the emerging Big Data vertical, arose from internet services computing, especially the MapReduce [11], Google File System [12], and BigTable [13] tools and their open source siblings, Hadoop [14], its distributed file system (HDFS) [15], and HBase [16], respectively. These systems were developed with a specific system model: identical cost-optimized nodes containing all the compute and storage available to the cluster, simplified semantics tailored to target applications, and the expectation of frequent failures [12]. With this heritage, interoperation with systems and tools from other environments such as high performance computing (HPC) can not be taken for granted, including traditional HPC storage. Because HPC computing systems are of comparable scale to Big Data clusters, it is particularly interesting to be able to support both types of applications using existing HPC storage for convenience and load sharing, if not consolidation for lower associated costs.

With the emergence of resource allocators like Mesos [17] and Yarn [18], a Big Data cluster can dynamically distribute resources between different parallel program schedulers such as Hadoop or HPC's ubiquitous Message Passing Interface

(MPI) tools [19]. This enables a sharing of clusters arising from the needs of internet services and those arising from the needs of high performance computing. Switching a set of nodes from executing Hadoop programs to executing MPI programs is easy; its just stopping and launching a set of user-level binaries. However, storage solutions for Big Data frameworks differ widely from that of traditional HPC. For example, HDFS stores write-once files that can only have one writing process, while HPC parallel file systems such as PVFS [20], Lustre [21], GPFS [22], and PanFS [23] support concurrent writes to the same file from thousands of processes. Further, HDFS was designed to store all data in the local disks of compute nodes, using replication for fault tolerance, and to interface to its servers through library (Java class) plugins. Parallel file systems, on the other hand, typically store all data in external storage systems, using RAID erasure coding for fault tolerance, and access servers through a Virtual File System (VFS) kernel module in each host. Therefore, if data is stored in the native format of one, it is not easily accessible to the other and copying is required between the storage mediums; with terabytes to petabytes of data the copy operation itself, not to mention the egregious amounts of wasted capacity, becomes prohibitively expensive.

2.1.1 The NAS and HPC Narrative

Nevertheless, due to the attractiveness of the solutions in the Big Data space, many organizations have acquired or put aside a separate set of nodes for exclusively Hadoop compute and storage and have suffered through the cost of capacity waste and expensive copies. Doing so may not be economically or tractably possible due to the huge datasets in traditional HPC NAS storage, but even if it were, the scattering of storage throughout the compute nodes comes with a number of drawbacks, which motivate our effort to seek consolidated compute for HPC and Big Data computing atop a NAS system:

- *Loss of Infrastructure Consolidation:* Unlike traditional POSIX filesystems, it is non-trivial to execute a variety of applications on HDFS without adapting them to its specific semantics and interface.
- *Forced Import/Export:* Sharing data between HDFS and traditional storage requires an import or export, wasting storage and network resources.

- *I/O Performance Degradation:* For more typical workloads I/O performance degrades since Hadoop is tuned for performance on large datasets.
- *Loss of High-Availability:* The Hadoop NameNode is a single point of failure and requires administrative intervention when downed.
- *No Modification of Files:* It is impossible to modify previously written data, as HDFS is a write-once-read-many distributed file system.
- *Inefficient Compute-Storage Coupling:* When an HDD fails, system administrators may be forced to power down the node or at least restart Hadoop services, resulting in loss of computational resources.

While we argue these reasons give credence to even considering NAS for use under MapReduce, we admit there are other approaches possible in exploring solutions to this problem, such as adapting Hadoop to more efficiently handle scientific problems as recent research has attempted [24, 25]. Therefore, it is important to pause here and clarify that we attack it from the very specific narrative that existing traditional HPC architecture is in-place already. Specifically, we make the following key assumptions: First, the HPC compute is already in place and tuned to efficiently operate with discrete, high-performance NAS storage. Second, the HPC applications developed over many years, which run on these systems have been written in a combination of C or Fortran and MPI to maximize performance for their extensive executions and presume POSIX or near-POSIX semantics, making wholesale transition to MapReduce and a pure Hadoop environment highly intractable. We argue these assumptions reflect the reality of most supercomputing centers in the world today.

This narrative and its assumptions are critical to point out so that it is clear experimentally comparing MapReduce atop NAS against traditional Hadoop installations would be both inappropriate and discordant with our theme – we are not attempting to prove traditional HPC nor a converged architecture is the way of the future for HPC. *Our goal in this work is to show, if you presume the existence of high-performance NAS storage and compute nodes with limited storage, that using a Big Data computation framework such as Hadoop atop this NAS storage is not only possible, but, if cleverly configured, can be efficient and thereby expedite time-to-science for post-execution analytics.*

2.1.2 Contributions

In this work we seek to explore possible architectures that allow Hadoop MapReduce to run alongside traditional POSIX applications and against a consolidated storage system provided via NAS. Our **first contribution** towards this effort is a thorough exploration of the following three architectural arrangements that use existing software solutions to accomplish the aforementioned goal:

1. *HDFS as a Client Service*: As HDFS employs node-local VFS to store chunks, reconfiguration can replace node-local storage with remote NAS storage.

2. *HDFS as a Wire Protocol*: Alternatively, because HDFS is a client-server model with a network protocol for all communication, it could also be treated as a Network-Attached Storage (NAS) protocol like NFS [26] or CIFS [27], with the server running within the NAS system itself.

3. *No HDFS*: While HDFS requires MapReduce applications to efficiently operate on its data, MapReduce will efficiently operate on non-HDFS data if configured correctly, so skipping HDFS entirely and going directly to NAS is possible.

In each of the above architectures we thoroughly analyze the impact the arrangement has on *reliability* and experimentally demonstrate the effect that each has on *application performance*. Finding the above solutions satisfactory yet suboptimal, our **second contribution** is the design and development of a new Hadoop FileSystem interface class. Note that Hadoop applications perform I/O via a FileSystem class which can be replaced. Further, alternative FileSystem implementations to HDFS are available for communicating to Amazon S3 [28], CloudStore [29] and PVFS [30].

Our novel file system, the *Replicating Array of Independent NAS File System (RainFS)*, overcomes all of the major issues we discovered. As we had done with the three more standard approaches above, we analyze and evaluate RainFS along the dimensions of reliability and performance. We experimentally demonstrate the performance advantages of RainFS to be as high as 127% for write-intensive workloads and 217% for read-intensive workloads when solely utilizing erasure-coding reliability mechanisms. This superiority continues when we combine replicating and erasure coding reliability mechanisms, demonstrating as high as 254% for

write-intensive workloads and 210% for read-intensive workloads. In all tests performed, RainFS performed as well or better than all other architectures tested, while providing reliability guarantees exceeding any of the architectures which perform comparably to it.

2.2 Background

2.2.1 Overview of HDFS

When Google introduced its MapReduce framework [11] in 2004, a restructuring of large scale data analysis was spurred with the most notable open-source implementation of Google's framework being Hadoop [14]. The Hadoop sub-projects we utilize are Hadoop MapReduce, Hadoop YARN, and the Hadoop Distributed File System (HDFS). MapReduce allows users to write parallel programs that are automatically broken into Map and Reduce tasks and executed in parallel within a distributed environment. YARN allows multiple resource management and scheduling mechanisms to co-exist (MapReduce, MPI, etc.) in one managed cluster. HDFS, taking its initial specification from the Google File System [12], makes distributed storage accessible to MapReduce programs, and is tuned to perform well for local hard-disk-drives (HDDs) in the same cluster as the computation. This architecture is now referred to as *Converged Storage*. Note that converged storage breaks from previous architectures that separated processing from storage, normally taking the form of commercial RDBMS or MPI atop NAS systems.

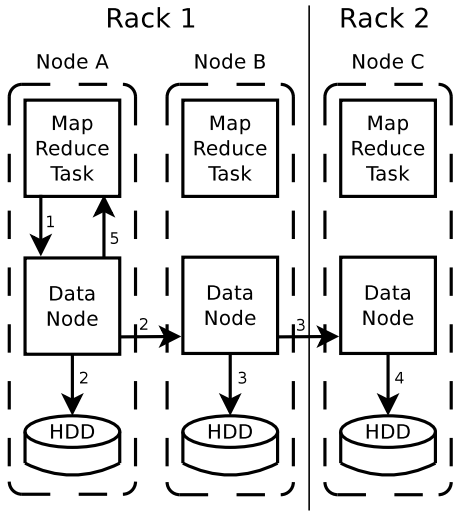


Figure 2.1: Flow of I/O from MapReduce tasks, through HDFS, and eventually to each nodes local HDD.

Hadoop MapReduce and HDFS have been shown to scale to thousands of nodes,

millions of files and petabytes of storage [31]. HDFS provides double-disk failure tolerance via file replication, out-of-band metadata access, is designed in Java for platform independence, supports Unix-style file permissions, and automatic capacity balancing between nodes. Hadoop has been adopted by many organizations, some of the most notable being Yahoo!, Facebook, Twitter, and LinkedIn.

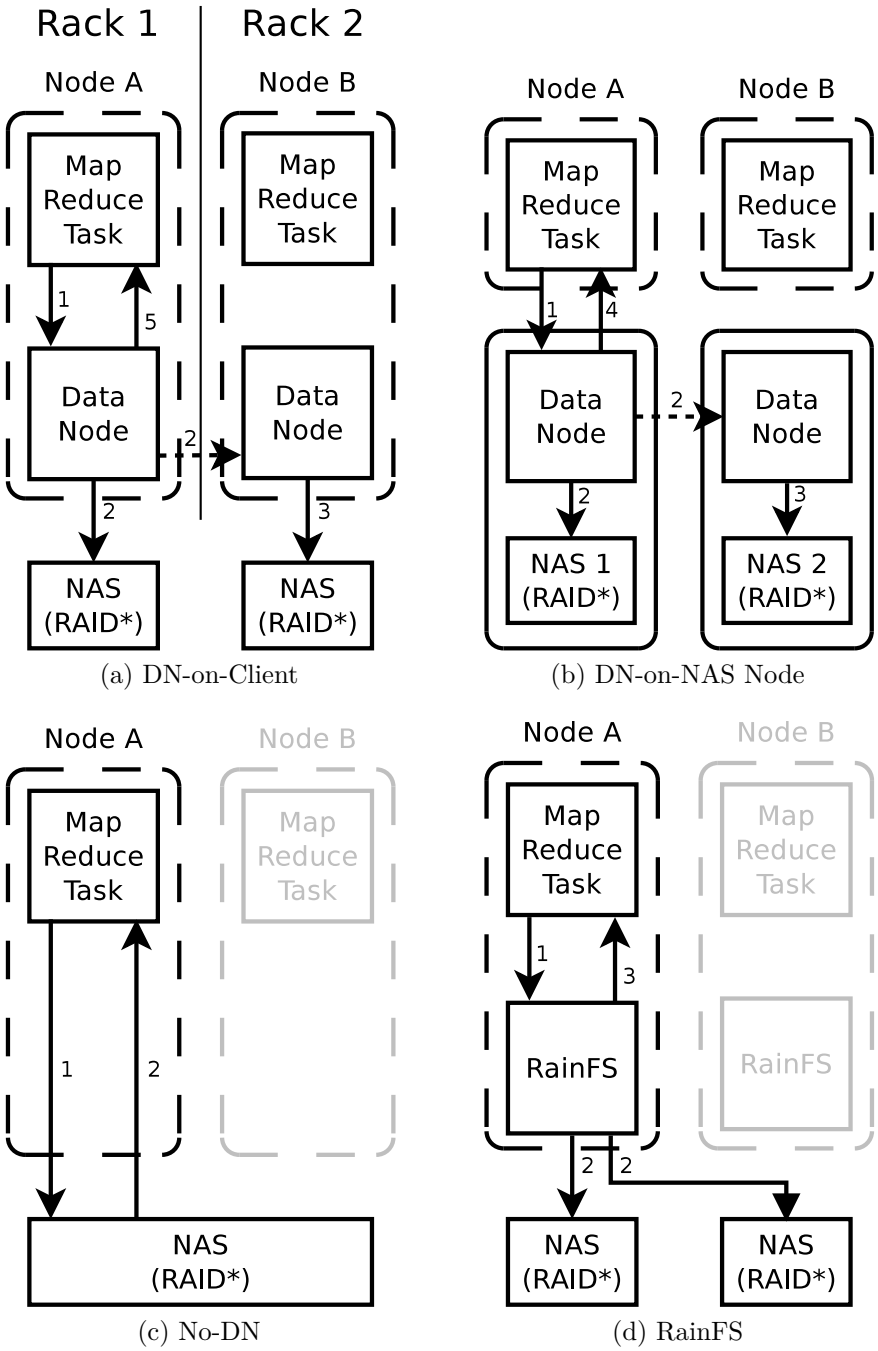
2.2.1.1 Replication in HDFS

HDFS achieves double-disk failure tolerance by replicating a file across multiple nodes (and therefore distinct HDDs). The process, shown in Figure 2.1, proceeds in the following manner: First, contact the NameNode for file creation (for clarity, communications with the NameNode are excluded from all diagrams). A response will provide the locations to write each copy and requesting Node A will begin concurrently writing its first copy to the local disk and its second copy over the network to Node B. The NameNode will attempt to specify a rack-local Node B to take advantage of higher-performance intra-rack communication, and specifies a rack-remote Node C to provide rack-failure tolerance. Node B concurrently writes to its local disk and pipelines that replica to the last destination, Node C, referred to as *replication pipelining* in HDFS. It is intended to share the replicating work with another node (B) and decrease the total time to replicate.

2.3 Architectures Explored

We evaluate three architectural options regarding where the DataNode (DN) daemons are located; we examine running them on the client node as in typical Hadoop usage, running them on the nodes within the NAS system (as if HDFS were a wire-protocol like NFS or CIFS), and bypassing DN daemons altogether, as shown in Figure 2.2.

The simplest manner in which one can utilize the Hadoop framework with NAS is to specify NAS mount points instead of paths to local storage. As can be seen in Figure 2.2a, this architecure is the most similar to the traditional HDFS setup shown in Figure 2.1 since the DN daemon still runs within a client node. Hadoop will attempt to write to and read from these paths and, finding that it can do so, will happily begin to use it as if it were a local drive. However, as a side-effect of HDFS believing this is a local drive, one will have to make sure to provide unique



(a) DN-on-Client

(b) DN-on-NAS Node

(c) No-DN

(d) RainFS

Figure 2.2: Flow of writes in the different architectures we explore, shown with a replication level of 2 for all but Figure 2.2c, which relies on internal redundancy. Dashed arrows signify communication that is purely network overhead, and grayed out nodes indicate that they have no role in the writing of the file from Node A.

	RAID 5	RAID 6	RAID 5	RAID 6	RAID 5	RAID 6
	Repl. 1	Repl. 1	Repl. 2	Repl. 2	Repl. 3	Repl. 3
DN-on-Client	1 / 0	2 / 0	3 / 1	5 / 1	– / –	– / –
DN-on-NAS Node	1 / 0	2 / 0	3 / 1	5 / 1	5 / 2	8 / 2
No HDFS	1 / 0	2 / 0	– / –	– / –	– / –	– / –
RainFS	1 / 0	2 / 0	3 / 1	5 / 1	5 / 2	8 / 2

Table 2.1: The number of concurrently failed disks or racks (shown in disk / rack format) that a given architecture can tolerate *without data loss*. Considered for all combinations of typical replication and RAID levels, and dashes (– / –) used to indicate an architecture cannot operate at this reliability level.

paths within the mount-folder for each node to use or else nodes may accidentally corrupt each other's files. As we later show, while easy to setup and get started with, this method has serious reliability and performance short-comings.

The second manner we explore using Hadoop on NAS is to move the DN out from each client and to run it directly on NAS nodes. In this architecture we specify, within the NAS nodes, paths to the mount-point(s) for the remaining NAS slaves so that incoming data to the node can be sent via these mounts to the individual storage nodes. This data flow is depicted in Figure 2.2b. However, forcing all of the data both for reads and writes through these nodes creates serious performance bottlenecks, as we experimentally demonstrate later.

Our third option is to completely avoid using HDFS, directing MapReduce tasks to access the underlying NAS mounts as if locally available to the system. One large caveat is demarcated in Figure 2.2c – the underlying NAS storage must either be a single system or multiple systems that provide federating services, such that the exposed namespace is unified and the NAS systems transparently achieve striping among themselves. This requirement is an artifact of MapReduce jobs being designed to operate on a single path, rather than a series of paths because HDFS is presumed to be in use. In order to fully remove HDFS, these NAS systems must expose a single namespace, provide reliability mechanisms, and provide load and capacity balancing.

2.4 Reliability Analysis

Reliability guarantees and the means by which these guarantees are kept vary between NAS systems and HDFS. In this work we consider the following fault model:

- **Failure of a Disk**: The disk is the most basic unit of storage, and is the most common part to fail.
- **Failure of a Rack**: Inability to get to an entire rack (e.g. top-of-rack network failure) or destruction of an entire rack of storage leads to the transient or permanent failure of an entire rack, and storage systems must provision for such failures.

To ease discussion, we use the term *failure domain* to refer to the range of physical resources that lose data upon failures greater than what it can tolerate. The trade-offs when considering the size of a failure domain tends to be a performance/reliability trade-off; large domains (across many NAS systems) allows for high-bandwidth access to single files since many disks can simultaneously help in the access, whereas small domains (perhaps one domain per NAS) reduces the chances of concurrent failure but have segmented namespaces and therefore fewer disks can help for a single file access.

2.4.1 Failure in NAS

To continue to operate upon failure of one or more disks in a NAS system, RAID (i.e. erasure coding) is typically employed, which can recover from a defined maximum number of concurrent disk failures based on the RAID level.

To handle failure of network and power components of the storage system that would otherwise result in inaccessibility of an entire rack, NAS systems employ redundant hardware (e.g., redundant network interface cards and power supply units).

2.4.2 Failure in Hadoop

HDFS provides tolerance to individual disk failure in a similar but nuanced manner when compared to traditional erasure coding done in NAS systems. In effect, HDFS

fault tolerance mimics declustered RAID1 with copies total. By replicating every file created on distinct HDDs, this assures that upon failure of a drive another copy will remain. The location and health of all replicas is managed by the NameNode.

To handle failures of an entire rack (caused by inaccessibility or physical damage) HDFS still employs replication, but it relies on knowledge of the topology of the HDDs to assure resilience. By assuring that at least one replica exists in a separate rack than the original copy, HDFS provides single-rack failure tolerance. The specific layout and flow of these replicas was diagrammed and can be referenced in Figure 2.1.

2.4.3 Combining the Architectures

We now consider how each of the proposed architectures placed atop NAS handle failure. When we report "failure tolerance of X disks", we are referring to the *maximum X which can be tolerated, no matter which specific X disks fail.*

Let us first consider placing the DN on the client node and configuring paths to NAS mounts. As mentioned, HDFS assumes each time it copies a file, it is in a totally separate disk (and therefore, failure domain), which will not be true if each client node sees paths to all NAS systems. When given multiple paths a DN will randomly select one of the paths for writing the file to load balance as this normally doesn't matter (typically each path is a discrete, local HDD). However, when DN-on-client points at NAS paths, we risk sending multiple replicas to the same NAS.

However, we discovered that achieving a replication level of two is safely possible by giving all of the clients in a given rack access solely to the single NAS system in the same rack (and thereby a single failure domain), and providing HDFS the topology of the system such that it knows all those clients are in the same rack. HDFS will therefore immediately attempt to make the second replica outside of the rack, assuring that the two copies are in two separate NAS systems. However, at triplication and beyond, Hadoop will attempt to create a rack-local second replica, and therefore duplication is the highest level of replication possible when using HDFS on NAS. This limits the possible reliability schemes as presented in Table 2.1.

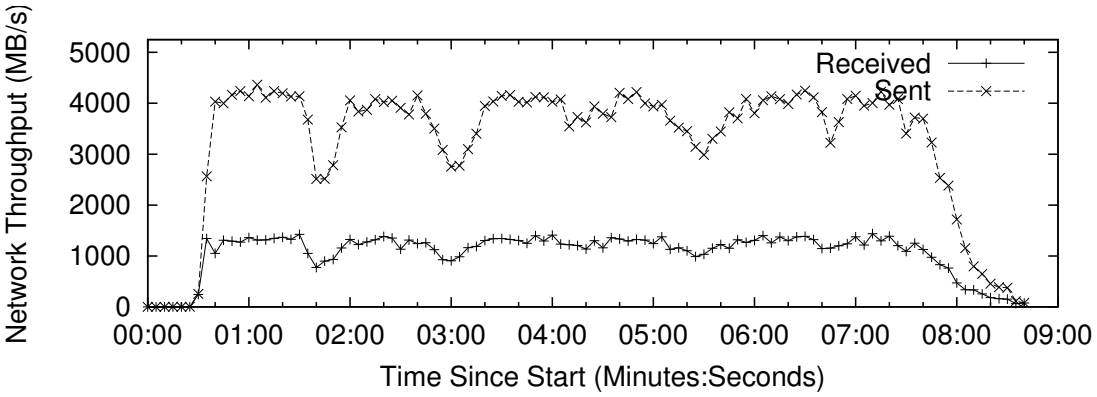
Moving to the DN on the NAS node architecture, one will see that this no

longer suffers from the replication level ceiling restriction as discovered with the last architecture. This is a result of placing a single DN on each NAS node; since we only have a single NAS system for each rack, there is a one-to-one mapping of DN to rack and therefore accidental duplication to the same rack becomes impossible. This allows for replication up to the number of NAS systems, achieving a comparatively wider range of reliability scenarios, shown in Table 2.1.

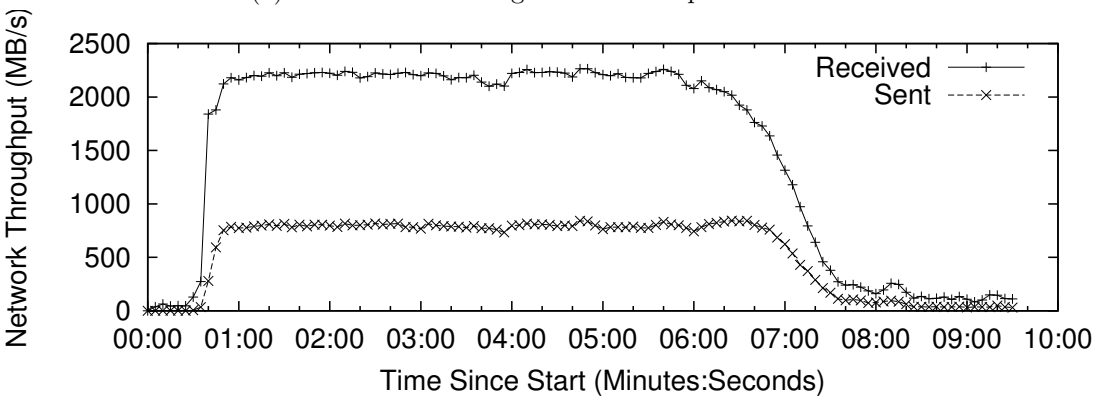
Last, examining the impact of guiding MapReduce to operate directly on the underlying NAS mount-point, we are faced with a much different situation since the NAS systems must be federated in order for this architecture to work. Because the NAS units are federated, RAID-5 will only provide single-disk tolerance across all of them, and RAID-6 similarly only provides double-disk tolerance. Further, without HDFS we lose replication, so we have no way to tolerate rack failure or ensure availability in the face of network, power, or some other fault in a rack that causes it to go offline.

2.4.4 Why Not Just NAS?

Finally, we recognize that many supercomputing systems may (and should) be architected to be reliable based on the guarantees provided by the parallel file systems and NAS alone. While this may be the case, and in which case utilizing the No-DN architecture may be the best choice, two potential use-cases exist for layering Hadoop replication on existing NAS RAID: First, doing so dramatically increases the reliability and up-time for files in an existing system that may only otherwise provide RAID5; higher reliability may be desired for the post-processing analytics so that chance of data loss for these final results is not only absolutely minimal, but also not tethered to the capabilities of the system to rebuild quickly. Second, layering Hadoop with replication on NAS also may make sense for smaller supercomputing centers whom may utilize NAS systems from multiple vendors. With such discrete namespaces, this would prevent the No-DN architecture from operating, and therefore layering Hadoop in-between would enable one to execute a MapReduce task to concurrently utilize all of the discrete NAS pools.



(a) Errant Pass-Through Write Transport Behavior



(b) Errant Pass-Through Read Transport Behavior

Figure 2.3: Write-intensive and read-intensive benchmarks on 50-node cluster and 5 NAS shelves, demonstrating poor write and read transport behaviors for DN-on-Client architecture.

2.5 Data Locality and Transport

In this section we consider the impact of locality changes when moving away from a traditional node-local Hadoop configuration to a remote storage solution.

2.5.1 Write Transport

Figure 2.2 provides an overview for each of the considered architectures on how writes flow from MapReduce tasks to NAS. When the DN runs on the clients as shown in Figure 2.2a, performance suffers significantly due to the errant network transits. Because HDFS assumes that it is working with individual HDDs, it believes there is no other way to get data to that HDD besides going over the network to the client that supposedly contains it. However, when all of these local

HDDs are replaced with paths to remote storage, the storage becomes equally close to all of the clients, and therefore an additional transit through some other node is complete overhead. It would be better for the DN in client Node A to write both replicas to separate NAS systems itself, but there is no way to effect this in HDFS without significant changes to its codebase that handles topology. This inefficient behavior is experimentally validated in Figure 2.3a, which shows that while writes should solely result in network send-bandwidth from the clients to the NAS, high receive-bandwidth is also occurring. In short, when we use HDFS with a replication level of 2 (original plus one copy) atop NAS, we should expect one out of three of the total network transits of the file to be overhead.

Moving the DN onto the NAS head nodes does not remove this errant pass-through behavior for writes, as shown in Figure 2.2b, but it does have the potential to alleviate it, depending on the relative link sizes available to nodes and connected to the NAS systems. Specifically, while the client machines in our experiments solely have a single Gigabit Ethernet link, each NAS system has two bonded 10 Gigabit links available. Nevertheless, it is quite computationally expensive to manage many tens, if not hundreds, of high-bandwidth, busy streams. This is a key issue we run into; even with only 10 clients per NAS system stream management overhead outweighs any benefits we gain from a higher theoretical peak performance.

In the third architecture, without HDFS shown in Figure 2.2c, there is no potential for any misunderstanding on how the storage is actually laid out – MapReduce is operating directly on normal files served via NAS. This enables all accesses to incur the fewest transits to maximize performance.

2.5.2 Read Transport

Reads seem considerably simpler than writes since no replication is performed – one imagines that reads should go directly through the local NAS mount and therefore avoid any pass-through situations. While this is true for our DN-on-NAS node and No-DN architectures, this intuition was found to fall flat for the simplest architecture of DN-on-Client.

We experimentally document the problem witnessed with our NAS on Client architecture in Figure 2.3b, which shows that while reads from NAS storage via HDFS should only show large amounts of received data, our per-node aggregation

shows significant amounts of sends as well. This phenomenon is the result of a task being placed on a client node that the HDFS NameNode does not believe to have the file that task operates on stored locally. Misplacement results in a request to one of the other nodes that the NameNode does believe to have the file locally, which starts a pull-through from the NAS system to the requestee and finally to the requester. These sends reach as high as around 33% of the bandwidth of the receives as shown in the middle of Figure 2.3b, indicating about a third of the tasks were misplaced.

2.6 RainFS

As we have shown, there are considerable overheads when utilizing HDFS to access NAS storage, yet bypassing HDFS entirely removes many of the reliability boons we had previously enjoyed. Further, bypassing HDFS requires that the NAS systems are capable of federation, which is not always the case, particularly with systems from different vendors. Therefore, we decided to implement a solution that attempts to avoid these overheads while concurrently retaining the ability to replicate over discrete failure domains and provide client-level federation. To that end, we present the Reliable Array of Independent NAS File System (RainFS), an intermediate file-system to replace HDFS for MapReduce on NAS applications.

2.6.1 Design Desirata

The goals in the design of RainFS were four-fold:

1. *Client-Level Federation of NAS Systems:* Enable MapReduce to take advantage of the performance of all of the available NAS systems concurrently *and* maintain discrete failure domains.

2. *Full Replication:* Restore the ability to replicate files written in MapReduce.

3. *No Data Pass/Pull-Throughs:* Neither writes nor reads should ever go through another client node on its way to or from the NAS systems.

4. *A Fair Namespace:* Create a framework-agnostic namespace where no imports or exports are required.

2.6.2 Implementation Overview

RainFS employs two key mechanisms to achieve the aforementioned goals. The first is utilizing symbolic links (symlinks) and hidden folders in order to allow a single NAS system to manage the remaining NAS systems and present a fair, federated namespace to MapReduce and POSIX applications. The second mechanism is providing metadata management via hidden metadata files beside the symlinks.

RainFS allows the NAS systems to be unfederated at the storage-level and yet, exposes a unified namespace, by maintaining that namespace on one of the many NAS systems available to the administrator. In that managing-NAS namespace, instead of storing the data directly when written from MapReduce, it stores symlinks that lead to the data files in one of a number of distributed, hidden directories. It is important to emphasize that the symlinks nor hidden metadata files are not on individual, local clients, but on a single managing-NAS system such that all clients see them in a unified manner. This enables concurrent reads on a set of files in the visible RainFS file system to be load-balanced across many or all of the NAS systems, depending on how many files are accessed at once. Further, this allows an MPI or other POSIX application to read a MapReduce-generated file directly from the symlink in the visible namespace of the managing NAS or to write its own set of files and allow MapReduce to turn around and process those files directly without import or export.

The second mechanism, metadata management via hidden files beside the symlink, allows RainFS to manage these many hidden files and folders that the symlink points to. As we will discuss in the following sections, there are numerous consistency issues that must be addressed on file create, delete, and move, and RainFS achieves all of this without its own centralized metadata manager.

2.6.3 File Operations

To enable easy adoption and code-reuse, RainFS extends from the abstract base class FileSystem as provided in the Hadoop code. This makes it a sibling to other FileSystem-extended classes like FTPFileSystem, where FTP servers are used for storage, S3FileSystem, where Amazon S3 is used as the backing store, and even HDFS, which also extends from the FileSystem class. Furthmore, for operations that do not require special handling of symlinks or hidden file metadata (e.g.,

a mkdir), RainFS re-uses code from the RawLocalFileSystem class to provide a similar range of operations available in other Hadoop filesystems. The operations which do starkly differ in RainFS from those available in RawLocalFileSystem are file creation, deletion, and move, and we detail them at length in this section.

2.6.3.1 Create

Algorithm 1 File Creation

```
1: procedure CREATE(filepath, replication)
2:     LOCK(FILEPATH)
3:     rndID ← RNDGEN.NEXTLONG
4:     nasStart ← rndID mod nasCount
5:     bucket ← rndID mod 2^bucketPower
6:     for i ← 0, replication − 1 do
7:         files[i] ← BUILDPATH((nasStart + i) mod nasCount, bucket, rndID)
8:         FILE.CREATE(files[i])
9:     end for
10:    CREATEMETASTORE(filepath)
11:    SYMLINK(filepath, files[0])
12:    UNLOCK(FILEPATH)
13: end procedure
```

Algorithm 1 creates a file by taking as parameters the path to the file and the replication level desired. First, we utilize NAS file locking to create and lock a hidden locking file beside the path of the symlink we are seeking to create. Since MapReduce is designed for Big Data (and therefore fewer, larger files), we do not believe a basic locking approach should have notable performance impacts on any realistic workloads. We then generate a psuedo-random long integer and identify both the first NAS system the symlink should point to as well as the *bucket* in use. Since the first NAS is randomly chosen and then subsequent replicas are just round-robined around available other NAS systems, the replicas will be uniformly distributed. We utilize the concept of *buckets*, which are truly just folders, and a configurable *bucketPower*, to ensure that it is unlikely that any single folder becomes overwhelmed with a huge number of files. Without buckets, depending on the NAS system in use, folders with a huge numbers of files could very well suffer performance degradation.

Then we create each replica, iterating through the available NAS systems as specified in the RainFS configuration file in round-robin fashion. Changing available

NAS systems is as simple as altering the configuration file, as it is read upon every call to the RainFS library; no restart required. However, in similar nature to HDFS files needing to be read and rewritten if the block size is desired to be changed, RainFS does not yet provide capacity rebalancing if the replication level changes. One will need to read, rewrite, and delete the old file to achieve rebalancing. Following creation of the replicas the symlink to the first NAS is created and the metadata information about the file is stored in a hidden file beside it.

Once the file is unlocked this procedure returns a new output stream for subsequent writes. Writes occur simultaneously to all replicas via threading, one thread per replica. This process is identical for subsequent writes sometime later after the file creation. On read access, only the data file that the symlink points to will be actually read from – reads are not performed concurrently from all replicas available. However, MapReduce programs frequently work with multiple files concurrently, so it should still achieve load balancing on read-intensive workloads across all NAS systems. In future work we are considering improving the read function so even single, huge files, are served from all available replica locations. This achieves client-level federation of the NAS systems, allows for replication all the way up to the number of NAS systems available, avoids transit overheads for both writes and reads since each client directly contacts the NAS systems for each of its I/Os, and maintains a namespace on the primary NAS system that non-Hadoop applications can access.

2.6.3.2 Delete

Algorithm 2 File Deletion

```
1: procedure DELETE(filepath)
2:     LOCK(FILEPATH)
3:     replicas ← GETREPLICAS(filepath)
4:     FILE.DELETE(filepath)
5:     for i ← 0, replication − 1 do
6:         FILE.DELETE(replicas[i])
7:     end for
8:     FILE.DELETE(filepath.metadata)
9:     UNLOCK(FILEPATH)
10: end procedure
```

File deletion as shown in Algorithm 2 works in nearly reverse order as create,

and is done so with good reason: if intermediate failure occurs partway through a creation or deletion, these routines are built to maintain a reasonably sane environment even in the face of failures. Further, partial creates or deletes should be easy to clean-up with this ordering by an independent, scanning RainFS checking daemon. In the delete routine the file destined for deletion is locked via NAS file locking as done with create and all of the replicas are determined from the metadata file. Then, the symlink is removed such that any subsequent operation should recognize that the left-over replicas should also be deleted. Finally, the replicas are removed followed by the metadata file. The metadata file is left until last in order to preserve information and expedite clean-up in the event of an interrupted delete.

2.6.3.3 Move

Algorithm 3 File Move

```
1: procedure MOVE(filepath, newpath)
2:     if filepath > newpath then
3:         LOCK(FILEPATH)
4:         LOCK(NEWPATH)
5:     else
6:         LOCK(NEWPATH)
7:         LOCK(FILEPATH)
8:     end if
9:     UPDATEMETASTORE(newpath, filepath)
10:    FILE.MOVE(filepath, newpath)
11:    FILE.DELETE(filepath.metadata)
12:    UNLOCK(FILEPATH)
13:    UNLOCK(NEWPATH)
14: end procedure
```

Last of the common file operations, file move, would be a tricky routine if it were not for locks provided via NAS file locking. As we mentioned, providing locks around the entire routine is a reasonable approach for creates and deletes – as a Big Data framework, MapReduce was never designed for rapid creation or deletion of huge numbers of tiny files. We find moves to also be satisfactory for such routine-encompassing locks because *we do not actually move the data, we only move the symlink and the metadata file.* This allows for a very short time to be spent in the locked section compared to a full data copy and subsequent deletion of the old data. Therefore, in the move routine, we lock both the target and the

source files in order to prevent any concurrency issues from arising. Once both are successfully locked, we first update the metadata file at the target and then and only then move the symlink to that target. Updating the metadata file at the target first prevents the possibility of a partial move resulting in lack of a metadata file alongside the new target symlink. Once the move is successful (metadata and symlink), we then delete the old metadata file at the source location and unlock both the source and target.

2.6.4 Failure Handling

The last consideration before experimentally comparing the performance of RainFS against the previous architectures is reliability of RainFS in the face of failure of a disk or rack.

First, just as in the other architectures, since it utilizes NAS storage, it gets single- or double-disk failure tolerance based on the RAID level the NAS system employs *for every NAS system*. However, unlike the architecture that simply bypasses HDFS, RainFS achieves federation at the client-level and therefore the fault domains among NAS systems are not conjoined. Concurrent failures in distinct NAS systems will therefore not aggregate – for five NAS systems and RAID-5, five disk failures can be tolerated without data loss in any of the failure domains if they occur in separate systems.

Furthermore, if an entire rack becomes unavailable and RainFS is unable to contact it for a read, it will simply iterate through the remaining replicas to attempt transfer from a NAS system on another rack. A notable caveat here is that the master NAS is a single point of failure; if it fails the unified namespace will be unavailable until it is restored.

Beyond what RainFS can promise in the face of failure or unavailability of one of the NAS systems, we must also briefly consider how consistency is maintained. We make one important assumption in this work to simplify RainFS and keep it out of the critical path for non-MapReduce applications: if a file is created via Hadoop, the user should take care to also delete it or move it with Hadoop. Users can of course (as it was one of our goals) read from those files using any application they wish, be it MapReduce-based or not. The reason behind this assumption is that since RainFS is solely in Hadoop, if an external user or application moves a

Component	Description	Per-VM
Processor	Intel Xeon E5240	2 Cores
Memory	DDR2 667 MHz ECC	3.8 GB
Hard Disk	Sata II 7,200 RPM	200 GB
NIC	Unknown Card	1 Gb/s

Table 2.2: Hardware and VM resources

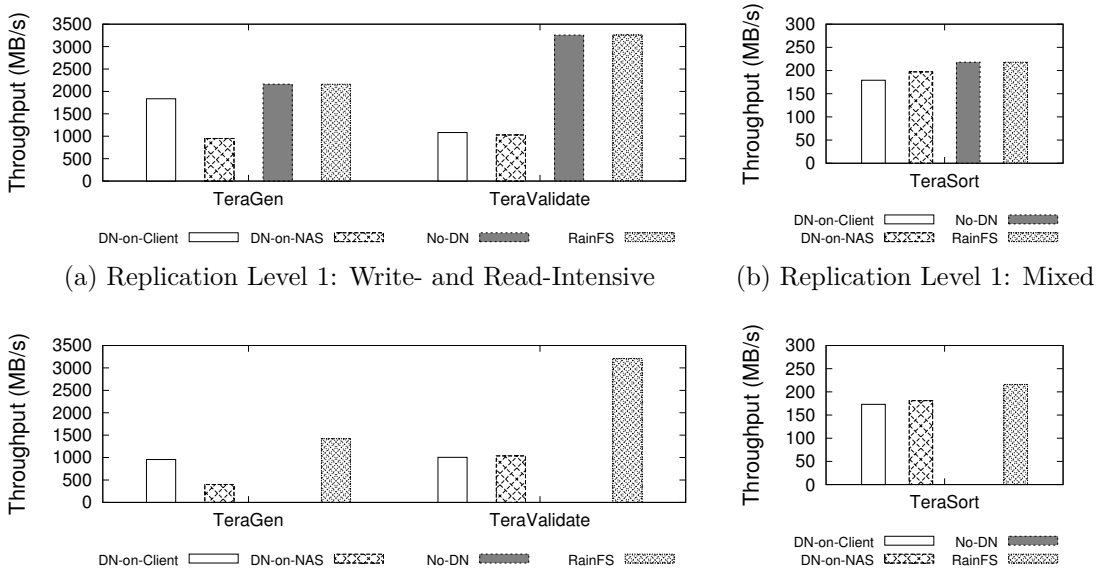
symlink in the unified namespace, RainFS will not be able to keep up and move the metadata file along with it. Similarly, external deletions of symlinks will leak storage since the metadata files and replicas still remain in hidden folders on the distributed NAS systems. This RainFS checker similarly attempts to complete or roll-back partial creates, deletes, and moves after a specified time-out. Improving the RainFS checker to be more robust beyond these capabilities is a target in future work.

2.7 Evaluation

We now describe our experimental environment, the benchmarks we used to tease out differences between architectures, and provide results for and discussion on our experiments.

2.7.1 Experimental Setup

To experimentally validate our expected overheads and proposed optimizations we used a medium-sized cluster of 50 nodes, which utilize five shelves of Panasas ActiveStor 12. The nodes have Gigabit NICs, which are connected to one of four Force10 S50n Gigabit switches. These switches are in turn connected by dual 10-Gigabit Ethernet uplinks to a single Force10 S4810p 10-Gigabit switch, which our NAS system is also connected to via two, 10 Gigabit Ethernet links bonded together per shelf. Each node is running KVM with a virtual machine image of CentOS 5.5, which is the environment for all of our experiments. Further specifics regarding the hardware is listed in Table 2.2. We required virtualization in order to take advantage of the Panasas DirectFlow client module, which at the start of this work was solely available for RedHat-based distributions and our hosts run Debian.



(a) Replication Level 1: Write- and Read-Intensive

(b) Replication Level 1: Mixed

(c) Replication Level 2: Write- and Read-Intensive

(d) Replication Level 2: Mixed

Figure 2.4: User-perceived throughput of read-, write-, and compute-intensive benchmarks on a medium size cluster using all four architectures. Shown for replication level of 1 and 2 (where 1 means only the original data file exists, and 2 means two copies exist within the NAS systems). The No-DN architecture, or where neither HDFS nor RainFS is used at all, cannot replicate and therefore is missing from the bottom graphs.

2.7.2 Benchmarks

Perhaps the most ubiquitous macro-benchmark in the Hadoop space is the TeraSort benchmark, which is a suite of MapReduce applications designed by Yahoo! in 2008 [32] designed to compete in (and enabled them to win) the terabyte sort competition [33] that year. The three major components of the benchmark are:

TeraGen: The first component of the suite is TeraGen, an application that utilizes MapReduce to automatically divide the work of generating a configurable number of rows of key/value pairs over available clients. This benchmark is almost exclusively write-intensive and therefore, is used in this work as our write-throughput microbenchmark.

TeraSort: The second component is the most complex and performs the sort itself. TeraSort incorporates a custom partitioning algorithm that uses a sorted list of $N - 1$ keys. This enables a simpler, nearly embarrassingly parallel sort that correspondingly scales well. It has a read- and CPU-intensive Map-phase, a

network- and memory-limited shuffle phase, which shares the now-partially-sorted data, and last performs a write-intensive reduce phase, where the data is merged and outputted to disk fully sorted. Because it exhibits a full spectrum of MapReduce application behavior, we consider this component of the suite representative of more compute-heavy applications.

TeraValidate: The last component of the TeraSort benchmark suite is the validation application, which simply reads through the entire set of data and makes sure each key is sorted properly (it is less than or equal to the one previous). This benchmark, barring any errors (we encountered none in any of our tests), is completely read-intensive and therefore is used as a read-throughput microbenchmark in this work.

This benchmark exhibits the behaviors we expect most HPC post-processing analytics will exhibit. For instance, it is easy to imagine a scientist would first simulate a nuclear reaction using traditional HPC compute and storage. Then, they might seek to leverage a big data framework such as MapReduce as explored in this work to do a light filtering of the data, which might generate an only slightly smaller dataset (like the write-heavy TeraGen). Then they may sort to find areas with the highest or lowest temperatures (like TeraSort), and last look through all of the sorted data to verify no anomalies exist (like TeraValidate).

2.7.3 Results

Having laid out our experimental framework and the Hadoop TeraSort benchmark suite we utilize for this work, we begin our discussion of results by first presenting a basic overview of the parameters we configured for the TeraSort runs. In these experiments we first write 0.5TB of data using TeraGen and then sort that data using TeraSort, which generates a separate 0.5TB of data. Finally, we read in and validate that the second 0.5TB dataset was truly sorted using TeraValidate. Additionally, for all of our runs, we execute each benchmark three times in order to find a reasonable average. With half a terabyte, each machine is writing at least twenty gigabytes of data over the course of the benchmark suite and reading the same amount, making this a considerably out-of-memory dataset. Nevertheless, we take precautions against cache-effects by flushing the cache on all of our client nodes.

As a last consideration, we utilized the local HDD in each machine as the temporary storage space for MapReduce during its shuffle phases. This decision was made after thorough testing of entirely NAS-based setups, where no local disks were in use for the benchmark. In those we found performance to be much better when keeping temporary data local to the machine itself rather than pushing it out over the network to remote storage and subsequently pulling it back when needed (often very soon after the push). We believe utilizing a single local disk for temporary shuffle data does not jeopardize the intent of our exploration for two reasons: First, no permanent data stays on that disk and thus the reliability guarantees that HDFS and/or NAS promise are not in danger. Second, while not all HPC systems have a local disk (some boot simply over the network), many do, including current #1 Top500 Tianhe-2 [34], and almost all others provide some form of solution for a fast scratch space even if it is remote.

Moving to the results, let us first consider the write- and read-intensive benchmarks for replication levels of 1 and 2 as shown in Figures 2.4a and 2.4c. In the former, three main take-aways surface: First, DN-on-NAS runs into significant performance degradation for writes – it only begins to compete with any of the other architectures on reads, and that is because the DN-on-Client architecture begins to suffer from poor task placement (resulting in data pull-through). Second, and related, the impact of this errant data pull-through phenomenon resurfaces for TeraValidate in the DN-on-Client architecture. While performance improves for the No-DN and RainFS architectures when going from writes to reads, performance plummets for reads on the DN-on-Client setup. Third, the No-DN and RainFS architectures perform almost identically in all tests. This is a result of both of these architectures being based off of the same code-base, which performs I/O almost directly through standard Java I/O libraries.

In the latter set of read- and write-intensive benchmarks performing duplication, we note two points of interest: First, the DN-on-NAS node architecture performs even worse than in the previous case, achieving less than 10% of the theoretical throughput available to the NAS systems. This seems to make a fairly cogent case against pigeon-holing a distributed data framework like Hadoop MapReduce through a limited number of master nodes; it simply will not scale or achieve higher replication levels well once those nodes are overwhelmed. Second, since the No-DN architecture, or the architecture that skips both HDFS and RainFS and

goes directly to NAS, cannot perform any replication, it is missing from this graph intentionally to accent the reliability/performance trade-offs.

Now examining the compute-intensive benchmark TeraSort as shown in Figures 2.4b and 2.4d, the findings are somewhat less striking but nevertheless fit intuition. First, because this benchmark is running on simply dual-core machines with limited main-memory, we should expect any improvements in storage access speed to only improve a limited fraction of the run time. This is demonstrated with much smaller swings from the worst to the best in the architectures. Further, DN-on-NAS node finally takes a win in this case because the DN responsibilities have been moved off of the compute nodes, allowing them to move faster, and I/O is not the bottleneck. Last, No-DN performs almost identically with RainFS for replication level of 1, and is excluded from replication level of 2 for the aforementioned reasons.

Lastly, we analyze this performance data in Figure 2.5 to determine how the various architectures fair when increasing replication level. As the simplest architecture matches up with, our intuition suggests if twice the amount of data is being written, then the slow-down should be two times. However, DN-on-NAS fairs worse than this number, coming in at 2.38 times slower and RainFS fairs better, showing only a 1.52X slowdown. In the former case, we believe this to be related to the already overburdened NAS node slowing down further as it continues to struggle with more high-throughput network streams. However, for RainFS we were initially quite unclear as to how it performed better than in the replication level 1 case. What we have found through microbenchmarks and careful observation of the test as it runs is that since RainFS makes use of threads to write both of the copies simultaneously when replicating, it is able to hide some of the overhead in the I/O path and the other stream of

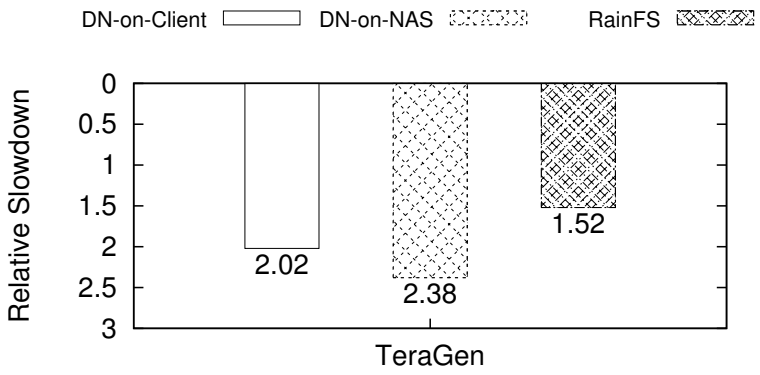


Figure 2.5: Throughput impact on write-intensive workloads when going from replication level of 1 to 2.

data fills in that space while the initial one idles.

2.8 Related Works

Despite the huge volume of work done in the cloud computation and parallel file system arenas, there are only two academic works to our knowledge that seek to find an efficient coexistence between them, and both are just short papers reporting research that is still in progress.

In the first work, MixApart [35], the authors create a new task scheduler and caching manager that relies on local HDDs to perform staging of data brought from shared storage. Their work does no exploratory research into whether standard incarnations of Hadoop are possible atop shared storage nor does it appear to operate without powerful compute-local storage, and therefore one of the major incentives for our work, infrastructure conslidation, becomes impossible. Last, there is no consideration of storage reliability or improving federation capabilities in their work; they rely on whatever the underlying shared storage can provide.

In the second work [36], the authors perform a comparative study of unmodified Hadoop MapReduce on HDFS versus a modified version of GPFS. Unlike our work, where we seek from the outset to use MapReduce on *function-specific dedicated storage*, this work attempts to retain the merged infrastructure of Hadoop where compute and storage share the same machines.

In the commercial space, EMC Isilon and NetApp have released products that perform, to one degree or another, actions similar to two of our explored architectures. In the former case, EMC Isilon provides their solution, OneFS [37], which exports a wire-protocol version of HDFS but that translates HDFS commands extensively into Isilon-specific data movement in the back-end. Our wire-protocol architecture (DN-on-NAS node) has parallels with this setup, although the exact implementation is not equivalent. In the latter case, NetApp provides their OpenSolution for Hadoop [38], which enables individual compute nodes in the Hadoop cluster with SAS-attached storage instead of their own commodity HDDs. This methodology has some parallels to our DN-on-Client setup, except we utilize NAS storage rather than directly-attached storage. These commercial implementations of architectures similar to ours were part of our motivation to explore this arena and try to bring some clarity about the benefits and pitfalls

of the various approaches to integrating Hadoop MapReduce and shared storage solutions.

2.9 Conclusion

As an increasing number of organizations and researchers in HPC begin to take stock of their I/O intensive workloads and consider a Hadoop environment, particularly for ad-hoc analytics and post-processing, we believe our work has shed light on the potential to combine new compute frameworks with traditional storage infrastructure. In this paper we have detailed the numerous reliability implications, locality impacts, and caveats involved in utilizing three different architectures to effect MapReduce atop NAS with standard software. We have further designed and presented RainFS, our custom Hadoop File System that works to overcome the many pitfalls observed in the previous architectures. Finally, we have compared these architectures along the dimensions of reliability and performance on a real cluster, and demonstrated performance improvements for RainFS as high as 127% for write-intensive workloads and 217% for read-intensive workloads for replication level 1, and as high as 254% for write-intensive workloads and 210% for read-intensive workloads when performing duplication to achieve higher reliability guarantees.

Chapter 3
Data Reduction: Scalable Deduplication and Compression Evaluation

3.1 Introduction

In the increasingly bandwidth-starved present and particularly, future, it is critical to extend our search for solutions to this problem beyond merely bringing all of the storage devices together. Specifically, in this chapter we explore the potential for reducing the space occupied by the data itself, without compromising the content in any way. When we discuss data reduction in this work we refer to the use of techniques to take the current representation of data, find patterns or redundancies in the data, and re-encode the data in a way that more cheaply (capacity-wise) expresses the original bits.

There are two well-known approaches to data reduction: compression and deduplication. NOTE: Some consider deduplication a compression technique. For sake of clarity, in this work we discuss them completely separately and beneath the broader umbrella of "data reduction."

Compression is a technique that attempts to search through data and build a dictionary of frequently-occuring sequences, and rewrite the data using the new dictionary. More frequently occuring sequences receive a symbol that occupies less bits, and less frequently occuring sequences receive more expensive symbols. For instance, the letter E, which has a relative frequency of 12.702% of the words in the

English language, stands a good chance of receiving a very low-bit count symbol to represent it in an English plaintext document. Numerous implementations of compression exist and are used on a daily basis, such as zip [39], gzip [40], bzip2 [41], cab [42], 7zip [43], and rar [44], and less commonly but more powerful, transparently compressing filesystems exist such as BTRFS [45], ZFS [46], and NTFS [47]. In this work we concentrate on the latter, transparently compressing filesystems, since that is the most readily applicable for the large data pools we focus on here.

Deduplication, on the other hand, performs data reduction at a much higher level. Instead of looking at a window of data within a single file to improve just that one, deduplication looks for relatively large sequences of data (many kilobytes) that occur multiple times within and across files in a file system. These redundancies are removed when detected, and only an origin reference to the data sequence is retained. All other copies of that data sequence merely maintain a pointer to the origin. As an example, whole files copied are a classic example where even the most basic form of deduplication will work marvelously, whereas compression would not work nearly as efficiently. While deduplicating file systems exist, such as ZFS [46], BTRFS [45], and LessFS [48], and commercial solutions such as NetApp's ONTAP Operating System [49], EMC Data Domain Deduplication Storage Systems [50], and Windows Server 2012 [51], because these solutions must be leveraged for the entire filesystem they are not nearly as commonly employed as compression.

While compression and deduplication techniques are well researched and implemented (particularly the latter), two large hurdles exist to properly implementing one or both on the large datasets. First, some datasets benefit from compression, some from deduplication, some from both, and a few benefit from neither. Blindly implementing both compression and deduplication on a dataset that is already bandwidth constrained stands a very good chance of further limiting its bandwidth since data now must be decompressed to be accessed and deduplication memory requirements can be severe. Therefore, it is critical to evaluate ones dataset on these techniques (and their many implementations and tunables) beforehand.

Unfortunately, particularly for large pools of data that would benefit in an absolute sense the most from these techniques, there is no freely available analysis toolkit available today that lets you know how much your data will benefit from compression or deduplication. While one might be able to guess at the intrinsic nature of his or her dataset and which technique it might benefit most from, for

larger pools of data this guessing-game becomes arduous and inaccurate. Current approaches require one to either enable compression or deduplication on your production filesystem (if it even supports it), or to copy the entire dataset onto a filesystem that supports compression and/or deduplication, and observe the overall change in capacity. The former may make the production dataset unavailable for sometime or extremely slow as the filesystem adapts to the change, either of which are rarely acceptable on production storage systems. With the latter approach, this becomes extremely expensive if not downright intractable: separate storage resources must be acquired, high-throughput network links must be established to copy the data in a reasonable fashion, and because of the tunables that impact the efficacy of both compression and deduplication, numerous iterations of copy and delete may be needed to reach the ideal setup. Moreover, one must commit to a specific vendor before performing a proper cost/benefit analysis on one's data to ascertain which compression and deduplication techniques best benefit the data at hand.

Therefore, in this work we seek to design, implement, and demonstrate the efficacy and accuracy of a scalable compression and deduplication evaluation toolkit named TreeChunks. We established three main design desirata for this work:

- In-Place Data Evaluation: The tool should be able to evaluate the data in-place, without any copies to another filesystem. Further, any temporary data generated should be small enough to reasonably fit on existing storage.
- Practical Scalability: It should be able to operate on large amounts of data in parallel and without memory constraints, and be able to utilize existing nodes connected to the storage system for processing rather than require specialized or tightly-coupled machines.
- FOSS and Well-Documented: TreeChunks should be Free and Open Source Software provided online and include detailed documentation along with examples to make adoption and use of the toolkit painless. Moreover, results should be anonymized and be able to be easily uploaded to a central website where knowledge about the impact compression and deduplication on various datasets can be aggregated for future research.

We demonstate in this work an implementation of these high-level design goals that allows one to evaluate existing data pools, even very large ones, at high rates

and in-place. This data reduction analysis toolkit, TreeChunks, adds a completely novel tool to the repetoire of system administrators and researchers to properly examine their datasets.

We layout this work in the following sections: First, we provide a more thorough background on compression and deduplication to aid the reader in understanding the differences, and then proceed to related works in the area that have furthered the development of compression and deduplication algorithms. Second, we detail more specific design goals we had for the TreeChunks toolkit, and enumerate the features the toolkit supports. Third and last, to provide examples of the tool, we use it on a medium-sized dataset replete with a wide-variety of file types to demonstrate the efficacy and features of TreeChunks.

3.2 Background

Compression and deduplication, while well-established techniques (especially in the case of the former), are rarely well-understood. Therefore, in this section, we provide further detail on how exactly compressors and deduplication algorithms work in order to aid the reader in understanding the specifics and differences between the two approaches, cite key papers in both categories, and list existing tools that explore these techniques and their differences from ours.

3.3 Compression

Compression tools such as Zip, Gzip, and Bzip2 provide compression and decompression facilities for Windows and *nix users. While different implementation-wise, they share a common history in the DEFLATE algorithm [52], which is comprised of both Huffman encoding [53] and LZ77 encoding [54].

Huffman encoding searches through a series of characters and replaces the most common ones with special bit representations. These "bit representations" can be termed symbols, and with Huffman encoding the most frequent ones are represented with the cheapest (i.e., lowest bit count) symbols, whereas the least frequent ones are replaced with the most expensive (i.e, highest bit count) symbols. For instance, given a string "AABBBCCCC" we can see the character C is the most frequent, B, is second-most frequent, and A is least frequent. Therefore, Huffman encoding will

build a symbol table where C requires the least bits and A the most. The string is then replaced with metadata including this table in the header of the string, and then the string itself. For this simplified example the table representation will outweigh the benefits, but for real files the benefits can be highly significant.

LZ77 encoding approaches compression from a different angle, recognizing that certain characters might have a repeating nature. Also known as "Run-length encoding," LZ77 encoding finds repetitions in the last N characters and replaces them with special codes indicating pattern and length. For instance, in the stream "Hello hello hello" the "ello h" component just following the capital "H" is the first repeating string. Therefore, when LZ77 encoding finishes with this string, it will produce something akin to: "Hello h[D=5,L=10]". The bracketed component specifies the distance back (D) the repeating sequence should start at, and the length said sequence should continue for (L) at the point of the component.

These descriptions and examples gloss over many of the details of powerful compressors, so, for more information, please reference [55]. While many compressors share the same heritage in Huffman and LZ77 encoding, resulting implementations vary wildly in terms of compression ratio, compression performance, and decompression performance [56]. As a very general rule of thumb, more complex and longer-running compression algorithms tend to compress files to smaller sizes and across a broader spectrum of file-types. Deciding whether to go with a high-throughput or high-compression algorithm (or a balanced algorithm or configuration) depends largely on the dataset and use-case at hand, which our tool sheds light on.

Furthermore, compressors and decompressors used in filesystems often vary from the per-file tools, enabling a richer set of mechanics. Examples of these include checking to see if the file fails to compress to a smaller size, and if it does not, falling back to traditional storage (so as to avoid the decompression cost) [57]. Also, maximum chunk sizes tend to be specified such that very large files (in the hundreds of MBs or GBs) need not be compressed in one expensive run (or decompressed and recompressed entirely upon change). Lower chunk sizes allow more flexible and cheaper manipulation of files without decompressing the entire thing, but can have a significant impact on overall compression ratio as shown in Figure 3.1. In this figure we explore how a plaintext file compresses when split into smaller and smaller chunks, beginning with full-file compression, and moving down to 1KB. For

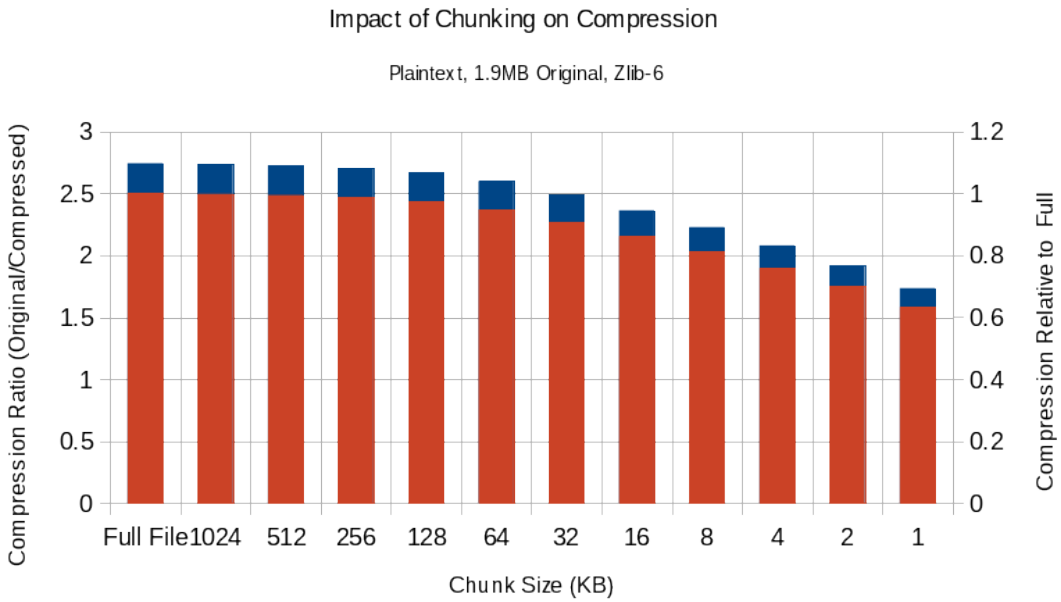


Figure 3.1: Demonstrating the impact of aggregate compression ratio when a text file (plaintext version of "The Brothers Karamazov" in this case) is chunked into smaller components. Full-file achieves best compression, but even at a 128KB chunk size near-optimal compression is achieved. Overlaid bar chart – blue is the absolute compression ratio, red is the relative compression ratio.

example, when compressing at the level of a single sector on newer HDDs (4KB), one forsakes approximately 20% efficiency in compression to achieve a far easier to decompress (and recompress) file.

3.4 Deduplication

The simplest form of deduplication is full-file deduplication, which Windows has supported in the form of Single Instance Storage (SIS) since 2000 [58]. This form of dedupe can be achieved extremely simply by storing just a lookup table with file-lengths and checksums, and upon match of both for a new file being written, perform a byte-by-byte comparison of the two and remove the duplicate. The largest issues with this tactic is that a) byte-by-byte comparisons are extremely expensive and b) even a single byte change in one of the files requires a full file copy (undeduplication). Nevertheless, researchers in [58] have argued that this simple tactic captures the majority of duplicated files compared to more complex approaches as we now

describe. Again, this is heavily use-case dependent – researchers in the former work are looking at a very specific data pool that happens to have numerous identical copies of files, which may not be common case.

To understand the more complex incarnations of deduplication that deduplicate at a sub-file granularity, it is critical to identify the two main issues involved: Duplication detection and file segmentation. The first issue seems simple – two lengths of data are the same if all of their bits match – but avoiding the need to read the file numerous times, which we have identified is expensive due to bandwidth bottlenecks, is key. For instance, in the above simple full-file deduplication we avoid blindly reading every file by using the heuristic that the file-length and checksum must match first. We only read a file which meets those criteria. If we could stomach the data-loss risk, we might be able to just say the files are identical if they have equal lengths and checksums. However, this is not safe enough, and therefore we cannot tolerate the risk.

Therefore, statistical approaches have been arrived at to cope with this, using cryptographic hashes over the data to provide extremely low probabilities of hash collision. The most commonly used hashes are MD5 [59], SHA-1 [60], and SHA-256 [61]. For the latter, SHA-256, which has the lowest chance of collision, given an exabyte of data and comparisons of every 8KB, we would encounter a collision about 1 in 10^{40} hashes [62]. Since a hard drive may read and return data that passes checks, but in reality has been corrupted, has a 1 in 10^{20} chance, it is widely accepted that SHA-256 provides more than enough safety. However, common practice (since most data pools are under an exabyte anyhow) is to adopt SHA-1 instead, which is much faster to calculate, since it provides 1 in 10^{20} chance of collision [62].

If we can accept that cryptographic hashes are good enough such that we no longer need to read a sub-file segment more than once and generate a hash for it that is used until changes are made, the next problem is deciding how to divide a file. In this work we term this "file-segmentation," or just segmentation, and the simplest approach is simply static segmentation. Static segmentation is the division of files every so many bytes (e.g., every 8KB). It is important to point out that small segment sizes are a double-edged sword: they concurrently allow for less affected segments on changes and better deduplication potential, but in turn require a much larger look-up table for the hashes compared to larger segments or

full-file hashes. This is particularly critical since lookup tables should ideally be in-memory. For example, 2KB segment sizes for 8TB of data without duplicates and 160-bit SHA-1 hashes will result in a lookup table reaching nearly 100GB, whereas moving to around 16KB requires a far more tractable 10GB of memory.

This solves the second problem from the simple full-file deduplication that small changes to a file will require full copies; now only the affected segments need to be copied and linked. However, what happens on a mid-file insertion? Since the bytes following the insertion all shift, this in turn results in distinct hashes for all subsequent segments, and indeed an insertion at the beginning of a large file results in a full file copy, which is needlessly expensive.

This brings us to a dynamic segmentation scheme that works to cope with these scenarios. Dynamic, or, "content-based" deduplication, must have the following qualities to be robust: we still must be able to control average size of segments, and it must deterministically choose split points. While numerous schemes exist, the one we'll cover in this work is called Rabin Segmentation, and is built upon research performed by Micahel O. Rabin in 1981, where he showed a new way to generate public keys [63]. In this work he uses randonly chosen, irreducible polynomials to fingerprint bit strings, which protects against unwanted changes being made to the data without knowing. This was repurposed as a content-based file segmentation scheme, and thus was born Rabin segmentation.

We lack space to properly address Rabin segmentation in this work, but a very high level version of the algorithm is as follows: First, slide a fixed size window (sized small, often 32-64 bytes) along the bit string and as each new byte enters the window, recalculate the Rabin hash. If the last N bytes of the window are zeros mark this as a segmentation point. The average segment size will be 2^N bytes, and usually minimum and maximum segment sizes are used for efficiency, which skew the average segment size a tad larger. This content-based deduplication solves the problem of insertions shifting all later bits because subsequent split points with the exception of segments very close to the insertion will remain the same, and therefore the hashes of those segments will remain the same. Only a very small window of bytes around the insertion will be copied and hashed as "new segments."

3.5 Design of TreeChunks

There are a number of design desirata we identified for TreeChunks to enable real users in the system administration and storage research and engineering spaces to best leverage it. These design goals can be grouped into three main categories: configurability, performance, and visualization. We break-down these categories as follows:

First and foremost, we wanted TreeChunks to be highly configurable, which means

- Enable/Disable Compressors/Deduplication Algorithms: Users should be able to turn off all compressors or deduplicators at-will, or decide which to test on their data at an individual level.
- Compressor Tunables: Internal compression configurations make a significant impact on performance and compression quality, so configuration decisions such as chunk sizes and compression levels should be allowed to be iterated over.
- Deduplication Tunables: Internal deduplication configuration choices such as average segment sizes, Rabin window sizes, and minimum and maximum segment sizes should all be selectable by the user.
- Responsive Performance Throttling: Since these executions may be running on machines needed for day-to-day operation and on storage systems similarly needed more during the day, administrators should be able to throttle the number of threads executing on a machine in real-time by just changing the configuration file.
- Granularity of Statistics Generation: Broad overview statistics will always be generated and presented, but more fine-grained and expensive statistics such as file extension frequency and volume should be able to be turned off.
- Enable/Disable Visulization: Graphs depicting directory structure and sizes will only be useful under certain conditions, and therefore, given how expensive it is to generate them, these should similarly be able to be turned off.

In TreeChunks we provide all of the above configuration features and describe them in more detail in the man file.

Second, because testing a variety of compressors over a spectrum of compressor tunables and deduplicators over a spectrum of deduplicator tunables rapidly becomes an expensive effort, we wanted to make sure absolute performance was achievable if the resources were available at hand:

- Scale Up: Since most machines today are multi-core, we wanted to assure even if a specific compressor did not support multi-threading, TreeChunks would keep as many processors busy as configured. This is achieved via lock-protected queues of files and permutations of compressors and deduplicators and tunables, a master thread which distributes work, and worker threads that take jobs.

- Scale Out: More even than multiple cores, we also wanted to enable multiple machines to be able to work on the data (provided the data is hosted in a centralized location such as a NAS) for processing as well as I/O performance. This is achieved by manual pre-division of the dataset right now, but dynamic auto-division is a planned future work.

- Single Read: Since bandwidth is such a limiting factor, we want to make sure once a file is read, all possible tests are performed on it possible before moving onto the next one (or even the next chunk). This is achieved via read-ahead buffering, and no re-reads are ever performed in TreeChunks.

In practice we have demonstrated performance efficacy of TreeChunks on 12 64-core machines, each connected via 10Gbps Ethernet to five shelves of Panasas ActiveStor 12. Because of the embarassingly parallel nature of TreeChunks it was able to push the storage subsystem to a peak I/O rate in excess of 5GB/s, and there was no clear indication that, given more machines, we couldn't have pushed the storage system to its theoretical limit of 7.5GB/s. It was still CPU-limited at that juncture. These performance characteristics give administrators and researchers the ability to, if they have the resources available, execute very heavy analyses over the weekends and evenings, and scale back the executions even in real-time during the weekdays so the compute machines and storage systems aren't affected for users.

Last, tree-based visualization of the impact each of these techniques and configurations has on the underlying file structure was a critical asset we wanted to add to TreeChunks:

- Graph and Node: The tree-based graph generated by TreeChunks visualizes subdirectories radiating out from the root folder, where edges are connections between parent and child directory, and nodes are directories themselves. No files are shown.
- Relative Size: The size of the directories *contents*, or immediate child files, is represented visually with varying sizes of nodes in the following categories: tiny (sub-KB), small (KBs), medium (MBs), large (GBs) and huge (TBs and up). All of these sizes are represented *before* any compression or deduplication is performed.
- Data Reduction Efficiency: To indicate how well the data in these directories are compressable or benefit from deduplication, we use white to indicate no improvement, cold colors (e.g., light blue, light purple) to indicate very small capacity savings, and hot colors to indicate high capacity savings. This provides a visually obvious categorization of subdirectories indicating which parts of the filesystem benefit most from different configurations of compression and deduplication.

This visualization is most-geared towards the system administrator who is using this tool to derive useful, reality-grounded knowledge from the dataset on his or her storage system. By having access to this type of a graph, he or she might be able to identify (the not-so-uncommon scenario) that most of the data under one arterial branch of the filesystem benefits greatly from compression whereas another does not, which might lead him or her to advise management that only enabling compression on that subdirectory is wise to achieve maximum benefit at minimal compressor overhead. Similarly, he or she might be able to identify that deduplication or compression is the clear winner, and have fruitful discussions with management that visually depicts why in a way that is connected to the filesystem's use by the organization.

TreeChunks is maintained at its website[1] and serves as a powerful tool researchers, engineers, and system administrators alike can use to extract knowledge

[1] www.ellisv3.com

that would be otherwise extremely costly to derive from datasets at their disposal.

3.6 Exemplary Evaluation

To highlight the aforementioned design goals and resulting features of TreeChunks we provide an exemplary evaluation of a medium-sized user dataset. This data is from a home-based NFS server containing a wide variety of files, including, but not limited to, operating system images in ISO format, backups in already compressed format, code in ASCII format, compiled code, large ASCII data files from experiments, music files in both FLAC and OGG formats, a large repository of academic papers in ASCII, PDF, and DVI format, various office documents in spreadsheet and ODF formats, and a large picture and video collection in JPEG and raw Canon image formats. This data occupies roughly 388GB of capacity as reported by the Unix "du" command. Because it is from a single personal collection, this data is not intended to be representative of the nature of any broader storage use-case, but due to the diversity of the data, breadth and depth of the directory structure, it serves well to illustrate the various features of the tool. Nevertheless, by allowing aggregation of results on the hosting website, we hope that in the near future a much broader spectrum of dataset analyses will be made public and broader conclusions about the nature of data and compression and/or deduplication's efficacy on said data can be arrived at.

As previously mentioned, TreeChunks is divided into two components, specifically, the Chunker and the Stacker. Chunker is the first component to be executed, and it is wholly written in C code since it does the majority of the computation. It is configured via a configuration file, usually "treechunks.cfg" in the same directory as the execution is begun. Please see the man file for more information on the configuration file syntax and features.

The master thread in Chunker first walks the entirety of the filesystem directory structure, and begins to build two queues. The first queue is a queue of subdirectories that still need to be traversed by the master thread in breadth-first fashion, and the second is the file queue that workers can pull files off of to work on. Once the file queue begins to be populated with files by the master thread, the worker threads immediately begin to lock, pull a file off of the queue for processing, and unlock the queue. Because of the far longer running nature of the processing of

the files compared to the lock, retrieve, and unlock process, simple mutexes more than suffice for this function.

Exemplary execution of Chunker is as follows:

```
$ ./chunker -v -i /mnt/store/ -o /mnt/store/.a_chunk_data/
========== START INPUTS ==========

General Inputs:
  start_dir:            /mnt/store/
  out_dir:              /mnt/store/.a_chunk_data/

Rate-Limiting Inputs
  threads:              8

Active Compressor Inputs
  chunk size:           16384
  chunk size:           131072
  zlib lvl:             3
  zlib lvl:             9
  lzo lvl:              1

Active Segmenter Inputs
  static seg size:              16384
  static seg size:              131072
  rabin_window:         48
  rabin_avg_seg:        16384
  rabin_min_seg:        8192
  rabin_max_seg:        32768
  rabin_window:         48
  rabin_avg_seg:        131072
  rabin_min_seg:        65536
  rabin_max_seg:        262144

========== END INPUTS ==========
Starting Up Thread ID: 1231275776
Starting Up Thread ID: 1239668480
Starting Up Thread ID: 1256453888
Starting Up Thread ID: 1248061184
Starting Up Thread ID: 1222883072
Starting Up Thread ID: 1140848384
Starting Up Thread ID: 1132455680
```

```
TreeWalk: Begin at 1399299740
path: /mnt/store//data
path: /mnt/store//sschoices
path: /mnt/store//panasas
path: /mnt/store//lost+found
path: /mnt/store//music
path: /mnt/store//audio
path: /mnt/store//archives
path: /mnt/store//backup
path: /mnt/store//switching_results.ods
path: /mnt/store//gallery
path: /mnt/store//documents
path: /mnt/store//code
path: /mnt/store//pictures

...output snipped...

TreeWalk: Complete, took 1252 seconds.
(757     /    168065) [1132455680] Processing file: /mnt/store//
    pictures/widener/DSCF1010.JPG...
(758     /    168065) [1256453888] Processing file: /mnt/store//
    pictures/widener/IMG_3538.JPG...
(759     /    168065) [1256453888] Processing file: /mnt/store//
    pictures/widener/FREEDOM 022.JPG...
(760     /    168065) [1132455680] Processing file: /mnt/store//
    pictures/widener/120.JPG...
(761     /    168065) [1256453888] Processing file: /mnt/store//
    pictures/widener/098.JPG...
(762     /    168065) [1132455680] Processing file: /mnt/store//
    pictures/widener/IMG_3507.JPG...
(763     /    168065) [1256453888] Processing file: /mnt/store//
    pictures/widener/IMG_3451.JPG...
(764     /    168065) [1132455680] Processing file: /mnt/store//
    pictures/widener/FREEDOM 024.JPG...
(765     /    168065) [1256453888] Processing file: /mnt/store//
    pictures/widener/DogsandFireworks 131.JPG...

...output snipped...

(168061 /        168065) [1231275776] Processing file: /mnt/store//
```

```
    app_data/mume/mmapper-2.1.0-source/build/src/CMakeFiles/mmapper.
    dir/proxy/moc_connectionlistener.cxx.o...
(168062 /       168065) [1256453888] Processing file: /mnt/store//
    app_data/mume/mmapper-2.1.0-source/build/share/icons/hicolor/32x32
    /apps/mmapper.png...
(168063 /       168065) [1256453888] Processing file: /mnt/store//
    app_data/mume/mmapper-2.1.0-source/build/share/icons/hicolor/16x16
    /apps/mmapper.png...
(168064 /       168065) [1231275776] Processing file: /mnt/store//
    app_data/mume/mmapper-2.1.0-source/build/share/icons/hicolor/48x48
    /apps/mmapper.png...
(168065 /       168065) [1256453888] Processing file: /mnt/store//
    app_data/mume/mmapper-2.1.0-source/build/share/icons/hicolor/128
    x128/apps/mmapper.png...

TreeChunk: Complete, took 13306 seconds to process 168065 files in
    4695 folders.
```

Input parameters for chunker here are -i for input or root directory to process all files beneath (that this user's account has permission to open and read), -o for output directory to store intermediate files, and -v for verbose mode (otherwise much of the output is suppressed). If no input is specified, the current directory will be used, and if no output is specified, /tmp/ will be used. Chunker begins by outputting the configuration file's parameters for user review and immediately starts up as many worker threads as specified therein. As you can see the worker threads have IDs that can be matched later on to the IDs in square brackets indicating which worker is working on which file. The master thread then begins to walk through the filesystem, adding directories yet to be explored to the directory queue, and files in need of processing to the file queue. This file tree walk continues concurrently as processing begins by the workers on the files, until sometime later (first output snipped statement) the walk is completed and only processing remains. At that time, time taken to walk the filesystem tree is shown, followed by the following information for each file processed: (1) Files processed over (2) total files to process, followed by (3) the thread ID working on that file, and last (4) the filename itself. Finally, much later when all processing of all files over all compression and deduplication algorithms has completed, the total time to process is outputted along with the number of files processed and the number of containing folders.

Intermediate files are outputted to the output directory specified by -i that will later be read by Stacker. The size of this intermediate data varies directly with the number and size of the configuration parameters specified in "treechunks.cfg." For instance, in this examplary case where a total of six permutations of compressors (two chunk sizes for ZLIB3, ZLIB9, and LZO1) are used for one medium and one large chunk size, and four deduplication permutations are used (two segment sizes for static and Rabin deduplication) for medium and large segment sizes, the resulting intermediate data occupies 5.3GB.

If one examines much smaller chunk/segment sizes the size of the intermediate files will be correspondingly much larger since one result of compression (before and after size) is stored for each chunk and a hash for each segment is stored for deduplication. Similarly, increasing the number of configuration options to search over expands the search space dramatically. For a robust, production use case, it would be advisable to search through two (small and large) or three (small, medium and large) chunk and segment sizes for multiple compression and deduplication algorithms. This will provide both insight into which algorithms work well on the dataset (compression versus deduplication, as well as compressor versus compressor and static deduplication versus content-based deduplication) as well as what costs are necessary to achieve real benefits (various sizes of chunks or segments as well as various levels of compression). Going beyond three sizes of chunks or segments or two or three compressors or deduplication algorithms will likely lead to diminishing returns in terms of knowledge gained on the dataset, and will take significantly longer to complete.

Once Chunker has completed, we move onto use of Stacker, which is written in Perl to leverage the statistics and visualization packages readily available in the language and because performance is less critical than with Chunker. Exemplary execution with resulting output is as follows:

```
$ perl stacker.pl --inDir=/mnt/store/.a_chunk_data/ --genGraphs --
    genExts
Aggregate Statistics (in Bytes Removed and Percent of Capacity
    Changed):
  Original Size:      414925739420
  Compression Statistics:
   For a Chunk Size of: 131072
    ZLIB (3) Compression:        316339601065     (-23.76%) (Time:
```

```
    15314.416823)
  ZLIB (9) Compression:        312154436694     (-24.77%) (Time:
    34696.987718)
  LZO (1) Compression:         334471845718     (-19.39%) (Time:
    669.832311)
 For a Chunk Size of: 16384
  ZLIB (3) Compression:        317351810106     (-23.52%) (Time:
    11626.952856)
  ZLIB (9) Compression:        313460120319     (-24.45%) (Time:
    18766.189788)
  LZO (1) Compression:         336420789399     (-18.92%) (Time:
    743.667518)
Dedup Statistics:
  Static (131072):             90382125434      (-21.78%) (Time:
    626.650498)
  Static (16384):              90819706742      (-21.89%) (Time:
    925.368151)
  Rabin (131072-65536-262144):          105333055911     (-25.39%) (
    Time: 8590.756560)
  Rabin (16384-8192-32768):    108771261920     (-26.21%) (Time:
    9964.059142)
```

In this execution of Stacker we opt to have both file extension analyses (–genExts) and visualization generation (–genGraphs) on the intermediate data generated by Chunker (–inDir=/mnt/store/.a_chunk_data). Stacker first generates overarching statistics about the dataset and the impact of each data reduction scheme and configuration on it. As we can see in these results, the total original size is expressed first, and then the small chunk size is examined for compressors. Here the compressor name and compressor level (shown in parentheses) are presented, followed by absolute bytes occupied after compression, relative percent capacity change, and finally aggregate time taken to compress. Looking at the larger chunk sizes and higher compression levels, we see the trend that longer times generate better savings, although the percent changes are small enough that it becomes questionable (for this dataset at least) how valuable the much increased time spent to compress is worth. Note that for datasets comprised entirely of uncompressable data, it is possible the relative percent capacity change could be positive. Further, please recognize that the timings shown in these statistics are approximations of computation cost – since real workloads are not available to TreeChunks in any

case, the costs to decompress or later, in deduplications case, the algorithms used for storing and retrieving data from the look-up table along with all the other complexities are unclear. Timings shown here are *solely for compression time and hash generation time.*

Following both chunk sizes for compression being shown, we see aggregate deduplication statistics. A similar format follows, the only difference being that the segment size is shown in parenthesis next to the algorithm name, and for Rabin the segment parameters read "(average-minimum-maximum)". Timings for deduplication are far less useful than for compression since a full deduplicating filesystem must handle a lot more than the mere hashing function.

If extension analyses are enabled as we did in this example, the following output will be displayed just following the aggregate statistics:

```
File Extension Statistics:
 Total Space Occupied by File Extension:
  JPG:          126653059844
  clean:        40825940001
  csv:          38102289874
  CR2:          36870071324
  avi:          35500168350
  trace:        25059971547
  jpg:          23727694292
  flac:         19519062681
  iso:          18794194944
  bz2:          9572835262
 Compressability [Bytes Removed / Total Bytes (Percent Change in
    Total Bytes Per Extension)]:
  For a Chunk Size of: 131072
   ZLIB (3) Compression:
    clean:      30117371144 / 40825940001 (-73.77%)
    csv:        28069289250 / 38102289874 (-73.67%)
    trace:      21460894295 / 25059971547 (-85.64%)
    h5:         4301070860 / 9402102223 (-45.75%)
    NONE:       2721213810 / 4047189814 (-67.24%)
    bz2:        2573559211 / 9572835262 (-26.88%)
    avi:        2511793156 / 35500168350 (-7.08%)
    CR2:        1959734652 / 36870071324 (-5.32%)
    iso:        1157064868 / 18794194944 (-6.16%)
    JPG:        795389501 / 126653059844 (-0.63%)
   ZLIB (9) Compression:
```

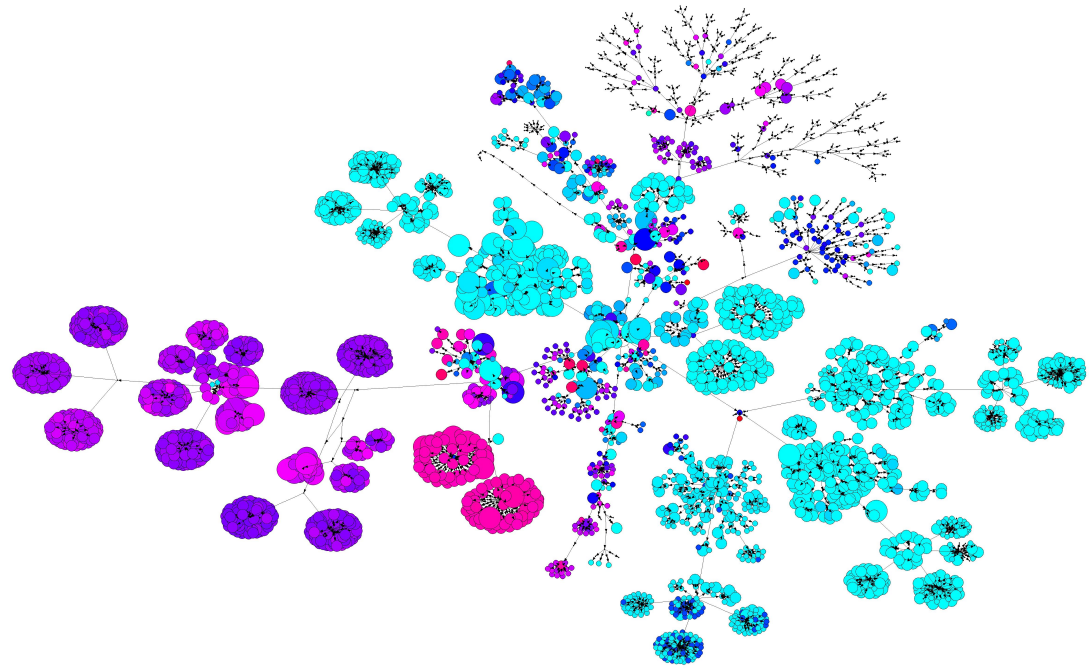
```
  clean:        31529370840 / 40825940001 (-77.23%)
  csv:          29417937487 / 38102289874 (-77.21%)
  trace:        22207863787 / 25059971547 (-88.62%)
  h5:           4610919416 / 9402102223 (-49.04%)
  NONE:         2852742061 / 4047189814 (-70.49%)
  bz2:          2664998885 / 9572835262 (-27.84%)
  avi:          2658372262 / 35500168350 (-7.49%)
  CR2:          1909668453 / 36870071324 (-5.18%)
  iso:          1198061991 / 18794194944 (-6.37%)
  JPG:          752202001 / 126653059844 (-0.59%)
 LZO (1) Compression:
  clean:        25279530027 / 40825940001 (-61.92%)
  csv:          23539497469 / 38102289874 (-61.78%)
  trace:        19575807968 / 25059971547 (-78.12%)
  h5:           3184950747 / 9402102223 (-33.87%)
  NONE:         2249197267 / 4047189814 (-55.57%)
  bz2:          2109755610 / 9572835262 (-22.04%)
  avi:          1050948796 / 35500168350 (-2.96%)
  CR2:          826377510 / 36870071324 (-2.24%)
  iso:          785521465 / 18794194944 (-4.18%)
  outv:         260413188 / 339442242 (-76.72%)
For a Chunk Size of: 16384
 ZLIB (3) Compression:
  clean:        30153981097 / 40825940001 (-73.86%)
  csv:          28030115014 / 38102289874 (-73.57%)
  trace:        21211982473 / 25059971547 (-84.64%)
  h5:           4278152472 / 9402102223 (-45.50%)
  NONE:         2704961218 / 4047189814 (-66.84%)
  bz2:          2459338594 / 9572835262 (-25.69%)
  avi:          2369892031 / 35500168350 (-6.68%)
  CR2:          1833004004 / 36870071324 (-4.97%)
  iso:          1089627053 / 18794194944 (-5.80%)
  JPG:          674969218 / 126653059844 (-0.53%)
 ZLIB (9) Compression:
  clean:        31410802007 / 40825940001 (-76.94%)
  csv:          29227014589 / 38102289874 (-76.71%)
  trace:        21872200544 / 25059971547 (-87.28%)
  h5:           4574507316 / 9402102223 (-48.65%)
  NONE:         2839598055 / 4047189814 (-70.16%)
  bz2:          2533513838 / 9572835262 (-26.47%)
  avi:          2518221904 / 35500168350 (-7.09%)
```

```
    CR2:        1851515225 / 36870071324 (-5.02%)
    iso:        1118642010 / 18794194944 (-5.95%)
    JPG:        685825141 / 126653059844 (-0.54%)
  LZO (1) Compression:
    clean:      24691313862 / 40825940001 (-60.48%)
    csv:        22967067736 / 38102289874 (-60.28%)
    trace:      19254303960 / 25059971547 (-76.83%)
    h5:         3121966982 / 9402102223 (-33.20%)
    NONE:       2213507845 / 4047189814 (-54.69%)
    bz2:        2032968901 / 9572835262 (-21.24%)
    avi:        938347249 / 35500168350 (-2.64%)
    CR2:        787755841 / 36870071324 (-2.14%)
    iso:        738968365 / 18794194944 (-3.93%)
    outv:       257528572 / 339442242 (-75.87%)
Dedupability [Bytes Removed / Total Bytes (Percent Change in Total
    Bytes Per Extension)]:
  Static Algorithm (131072):
    JPG:        61113438097 / 126653059844 (-48.25%)
    jpg:        12284551570 / 23727694292 (-51.77%)
    clean:      7928368537 / 40825940001 (-19.42%)
    csv:        5106964730 / 38102289874 (-13.40%)
    bz2:        2401158565 / 9572835262 (-25.08%)
    avi:        505937920 / 35500168350 (-1.43%)
    out:        419430400 / 632512191 (-66.31%)
    AVI:        158119972 / 5092443734 (-3.10%)
    mp3:        151507650 / 2925132418 (-5.18%)
    pdf:        148408669 / 2627155324 (-5.65%)
  Static Algorithm (16384):
    JPG:        61137391509 / 126653059844 (-48.27%)
    jpg:        12286943636 / 23727694292 (-51.78%)
    clean:      7948881305 / 40825940001 (-19.47%)
    csv:        5116762362 / 38102289874 (-13.43%)
    bz2:        2401158565 / 9572835262 (-25.08%)
    avi:        512765952 / 35500168350 (-1.44%)
    out:        419430400 / 632512191 (-66.31%)
    pdf:        176818525 / 2627155324 (-6.73%)
    mp3:        164549314 / 2925132418 (-5.63%)
    AVI:        158119972 / 5092443734 (-3.10%)
  Rabin Algorithm (131072-65536-262144):
    JPG:        64579401173 / 126653059844 (-50.99%)
    jpg:        12368723861 / 23727694292 (-52.13%)
```

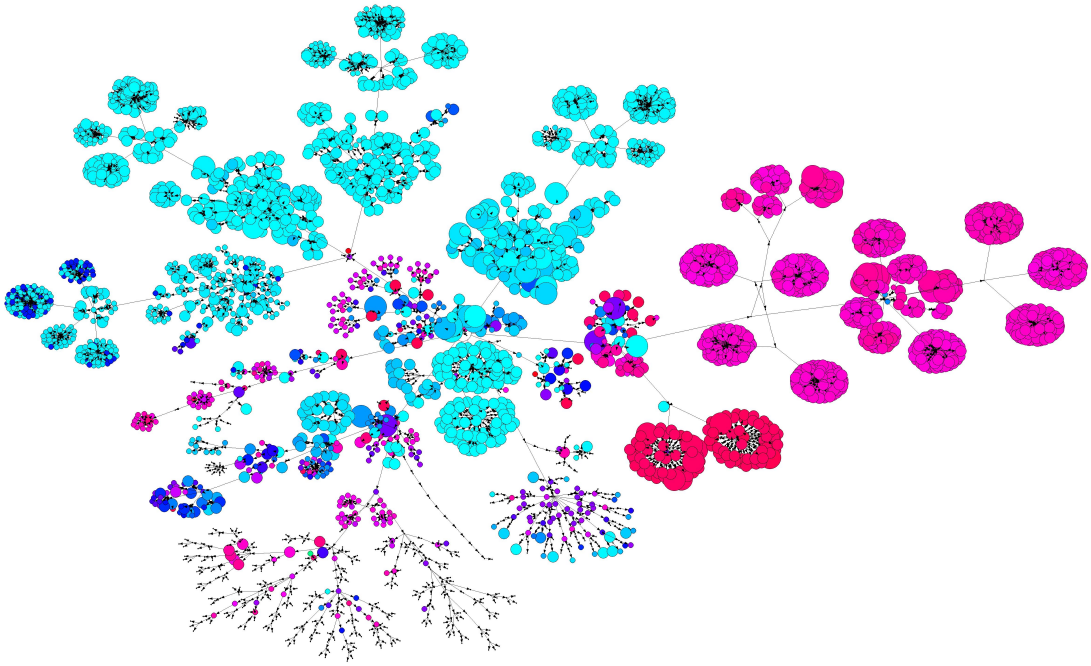
```
  clean:        11863904606 / 40825940001 (-29.06%)
  csv:          5930557821 / 38102289874 (-15.56%)
  avi:          4338238013 / 35500168350 (-12.22%)
  iso:          2619268144 / 18794194944 (-13.94%)
  bz2:          2401158565 / 9572835262 (-25.08%)
  out:          419430400 / 632512191 (-66.31%)
  mp3:          283609465 / 2925132418 (-9.70%)
  pdf:          198932206 / 2627155324 (-7.57%)
Rabin Algorithm (16384-8192-32768):
  JPG:          64856115993 / 126653059844 (-51.21%)
  clean:        12978095778 / 40825940001 (-31.79%)
  jpg:          12388994869 / 23727694292 (-52.21%)
  csv:          6436248574 / 38102289874 (-16.89%)
  avi:          4350427360 / 35500168350 (-12.25%)
  iso:          3182163618 / 18794194944 (-16.93%)
  bz2:          2401158565 / 9572835262 (-25.08%)
  pdf:          712011889 / 2627155324 (-27.10%)
  out:          419430400 / 632512191 (-66.31%)
  mp3:          326729389 / 2925132418 (-11.17%)
```

In the case of a full execution, all file extensions will be shown for completeness, not just the top ten. These results indicate that the dominating three file types in this dataset are JPG, clean, and csv (the latter two are ASCII data files), but although these take up the most space they are not necessarily the ones which benefit most from compression and deduplication. For instance, while clean and csv benefit massively from compression (varying from 60% to 77% smaller) JPG, an already pre-compressed format, saves less than 1% of its total capacity. These file extension results expand on the knowledge gained at a general level from the aggregate statistics by informing the administrator or researcher as to the specific use and nature of the dataset, and what types of its use will be impacted and improved by compression and deduplication.

Last, if visualization is enabled a geography of the impact of compression and deduplication is made available to the user. These graphs are generated in JPEG format and are saved into the intermediate data folder. One graph from each algorithm, LZO, ZLIB, static deduplication and content-based deduplication, are shown in Figures 3.2a, 3.2b, 3.3a, and 3.3b, respectively. As we can see, while compression impacts a much broader number of folders, it still benefits more in certain directories (e.g., the ASCII data file directories, compared to the JPEG

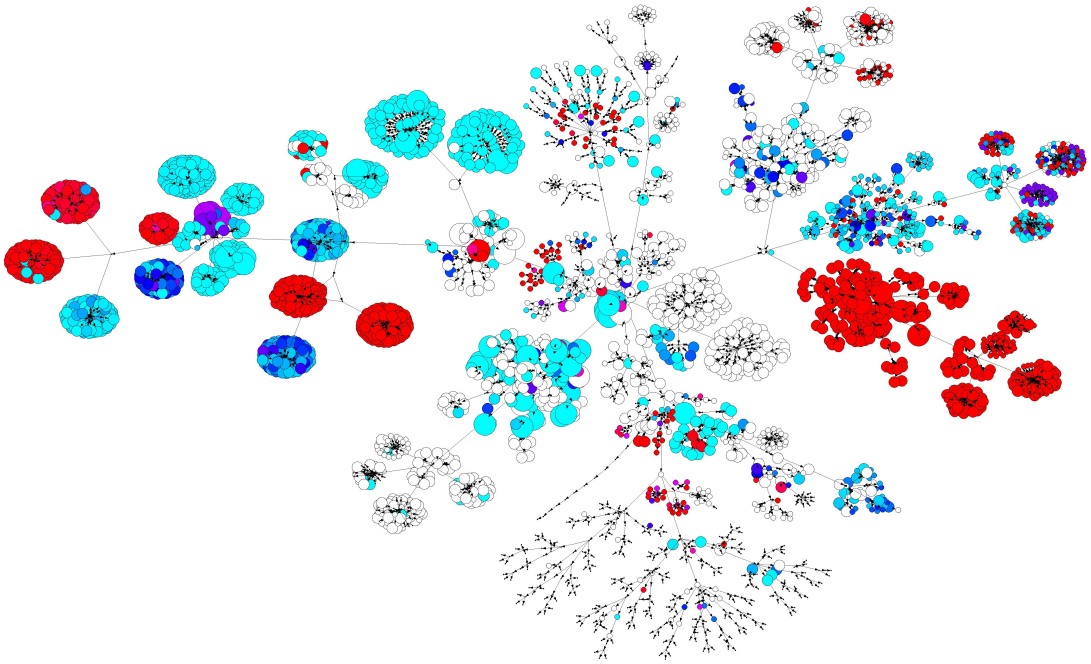


(a) LZO (1)

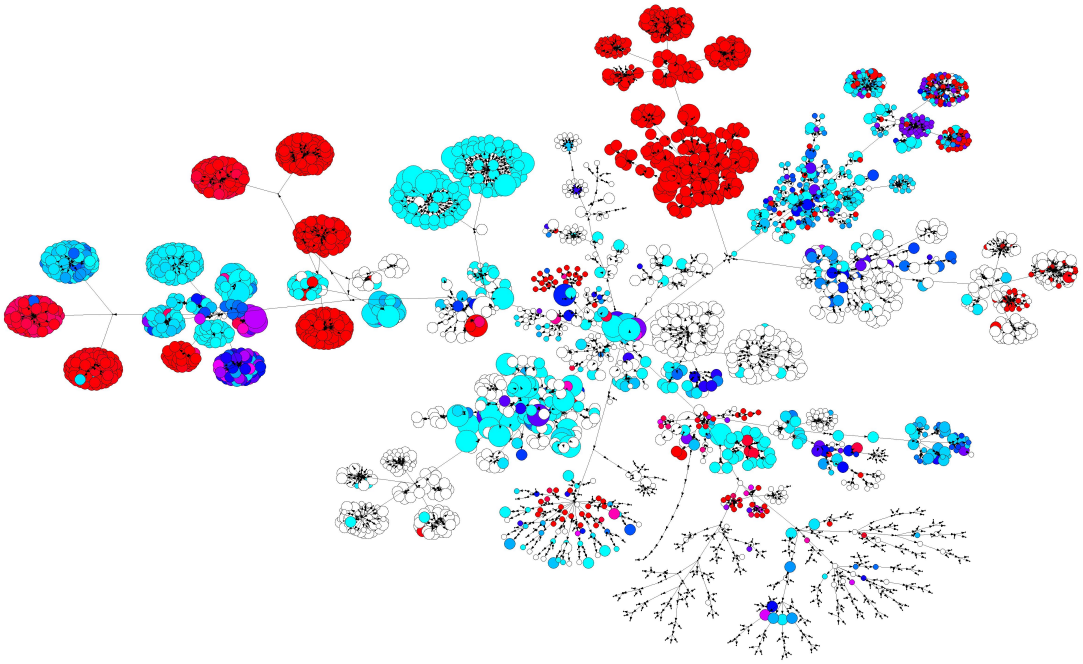


(b) ZLIB (9)

Figure 3.2: Compression efficacy filesystem map



(a) Static



(b) Rabin

Figure 3.3: Deduplication efficacy filesystem map

pictures directories) than others. Deduplication, on the other hand, is very polarized, often completely or close to completely eliminating the data in a given directory or not reducing it at all (i.e., large clusters of all red or all white). This is expected given the nature of deduplication, particularly for the dataset at hand from a single (organized and storage-space sensitive) user. Were multiple user's data examined who all worked at the same company, copied files or copied and very slightly changed files would be much more common. Nevertheless, this provides the user insight into which techniques work better, compression in this case, and which directories to enable it on if the filesystem allows directory-specific compression enabling/disabling (i.e., as available in BTRFS). By picking and choosing which directories to enable compression, the directories, such as pictures and music, which do not benefit whatsoever from compression, won't suffer unnecessary and costly compression attempts in the process.

3.7 Conclusion

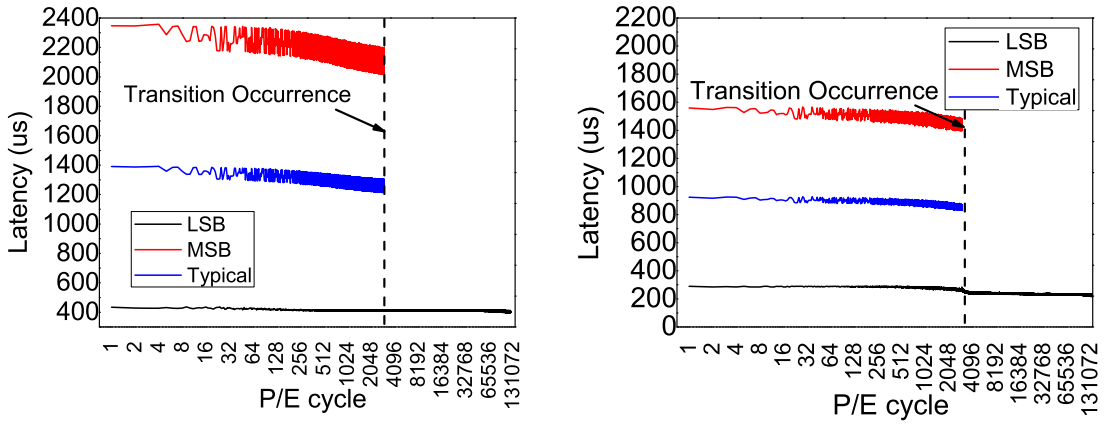
In this work we have erected design desirata for and implemented TreeChunks, a data reduction evaluation toolkit. As our evaluation demonstrates, TreeChunks makes a new tool available for researchers, engineers, and administrators alike to examine their data stores in more detail than they were ever able to short of entirely migrating to a new storage solution. TreeChunks provides full control to examine the impact of various compression and deduplication schemes and to test different tunables within each scheme in an automated fashion. It provides aggregate statistics demonstrating which scheme provides the most data reduction over the entire dataset, and allows the user to drill-down further with detailed information such as capacity consumed by file type, data reduced by file type, and visual topologies of the directory tree. These topologies give a quick but rich reference for where most of the capacity is located in on'es filesystem as well as how such capacity is impacted by various schemes. Ultimately, we believe this tool will enable storage experts of all types to explore avenues for data reduction that were previously intractably difficult, which should help alleviate the bandwidth limitation we are concentrating on in this thesis.

Chapter 4
Data Storage Improvement: Extending SSD Longevity

4.1 Introduction

Today, NAND-flash-based Solid State Drives (SSDs) are well-known and well-used in both commodity and commercial spaces. They bring to bear latencies and bandwidths far superior to traditional Hard-Disk Drives (HDDs), which helps to lessen the otherwise vast gap between main memory latencies and HDDs. Unfortunately, while SSDs have made great strides in the recent past, they still suffer from wear-out characteristics of the underlying NAND flash material.

Specifically, NAND flash is only able to handle a certain number of Program and Erase (P/E) cycles before a signal can no longer be stored stably. This issue is exacerbated by the reality that, to enable increased data density at a given feature size, techniques have been developed to store two, three, and even more bits into a single NAND flash cell, leading to major reductions in the threshold at which that cell must be declared "dead." [64, 65] Furthermore, new material science has led to reductions in the very size of such cells, which also impacts their ability to reliably retain data. [65] Because of these issues, as NAND-flash-based SSDs are used and age, their performance degrades as cells become harder to program reliably, and some cells die altogether and must be worked around. There are a number of works that have attempted to cope with these wear-out problems via techniques such as improved garabage collection to reduce write-amplification (the increased impact of a single write due to garbage collection), which we elaborate on in Section 4.5.



(a) MLC NAND Flash Drive A

(b) MLC NAND Flash Drive B

Figure 4.1: Proof-of-concept: Zombification process of NAND flash in two manufacturer's raw flash chips, demonstrating longer life and improved latency. Latency is shown to write all LSBs or MSBs in a page, and this process of writing all LSBs and then all MSBs in each page, for all pages in a block, and then erasing the block, are shown up until the block is declared dead. Typical latency is the average of the two.

However, in this work, rather than proposing yet another technique to reduce write-amplification or avoid temporary writes ever making it to the flash, we seek an entirely "unnatural" solution to the problem: we allow the cells to die a normal death, and subsequently bring them back from the dead, which we call zombified NAND-flash-cells. More specifically, upon the death of a cell, the point at which it can no longer reliably store two or three bits, we propose an algorithm that reduces the bit-capacity of a given cell by one. This has the curious side-effect of increasing the read, write, and erase speeds of a zombified flash-cell, and, on the whole, in many cases gradually improves the performance of the drive as it ages. To fully leverage these effects, we propose a novel adaptation to existing wear-leveling and garbage collection that greatly increases the potency of the ZombieNAND technique.

Nevertheless, there seems to be one major drawback to zombifying your NAND-flash cells: one bit of capacity is lost upon every transition to a lower bit-state. However, we argue that, since we are waiting until the cell dies to transition, the only alternative is to let the cell die and sit dormant, as it does now, which would lose you two or three bits for MLC or TLC cells, respectively. So, we prefer to think of it as *one or two bits are gained relative to complete death* – our results

demonstrate this to be the case with vastly increased drive lifetimes compared to traditional SSDs.

It is critical to note and understand why we, in absolutely no case, convert some of the flash to a lower state prior to their natural death. Manufacturers aggressively compete regarding how many P/E cycles their flash promises prior to death, and therefore, are extremely unlikely to implement any strategy that jeopardizes those guarantees. Therefore, towards the goal of developing a novel lifetime-extending scheme *that can be implemented in the real-world*, we make sure never to convert a cell until it has fully worn-out at that bit-level.

Before we begin, we need to motivate that this "zombification" is indeed achievable in *real* NAND-flash. Therefore, we performed and present results from an initial evaluation on two raw MLC NAND devices from distinct manufacturers in Figure 4.1. Therein we plot latency to write the least-significant bit (LSB) and the most-significant bit (MSB), the two bits in an MLC cell, over their lifetime. Latency shown is to write all LSBs or MSBs in a page, the lowest granularity of writes that can be performed. We continue the process by writing LSB and then MSB for all pages in a block, and then perform an erase on that block. This process is repeated until the block throws errors as demarcated with the vertical dotted line, at which point we transition from MLC to SLC and solely write to the LSB going forward. As can be seen, the cells continue to function in SLC-mode post-transition from approximately 3,000 Program/Erase (P/E) cycles all the way to in excess of 100,000 P/E cycles, and further, post-transition average latencies inherit the lower latencies of SLC NAND-flash.

However, manual transition of cells in this way is only a proof-of-concept – not a production-ready contribution. Therefore, to master the art of auto-zombification and management of zombie versus living NAND cells, we present the major contributions of this paper:

- **Physics-Accurate SSD Simulator:** We need to simulate much of the functionality of a flash-based SSD's flash translation layer (FTL), but, with the fidelity of a physics-based flash cell simulator because block-erase counters simply do not suffice (we do not know how much is really left post-transition). Therefore, we made extensive changes to a popularly used SSD simulator to enable it to perform physics-accurate transitions between bit-states. This is, to the best of our knowledge, the first work to enable accurate simulation

of the entire lifetime of an SSD from prestine state to its death, which we leverage extensively in our exploration of the ZombieNAND technique.

- **Controlled Wear-Unleveling:** We find that traditional wear-leveling results in too great a number of the cells dying at around the same time, limiting the impact of zombification. Therefore, we develop a novel wear-leveling mechanism that is both simple and effective at achieving a controlled degree of wear-unleveling such that a configurable number of cells transition (die and become zombies) earlier than the rest, which often improves the lifetime and the performance of the drive on the whole in advanced ages.

- **Industry-Relevant Results:** Changing the bit state sacrifices capacity and therefore has potential to cause the drive to use all reserved space and die earlier than as promised by the manufacturer. To avoid this and assure our results are industry-relevant, we set an invariant that we *never* may transition a cell to a lower bit-state unless it has reached the promised number of erases by the manufacturer. Further, by prioritizing simple but effective adaptations to existing garbage-collection and wear-leveling techniques over complex and fragile algorithms, we argue ZombieNAND can be implemented with only a modicum of difficulty across a wide variety of existing FTLs.

- **Expansive Environment Evaluation:** We witness a broad spectrum of performance and lifetime impacts based on varying environment conditions such as: proportion of reserved space, workload read to write ratios, working set size relative to reserved space size, and starting at MLC versus TLC. Therefore, in our evaluation, we use our simulator to perform an expansive search over all such environment conditions from prestine state to death of the SSD to provide conclusive evidence to when zombification can bring large benefits and when it cannot.

In this work we explore our ZombieNAND technique using both synthetic workloads and real traces. We demonstrate that ZombieNAND *never* hurts the lifetime of the SSD when compared against an SSD without it for over five-hundred different synthetic experiments and over sixty trace-driven runs. For workloads and configurations where it does help, which are the majority of the results presented, for TLC drives it can extend the lifetime from 20% to over 11 times for traces,

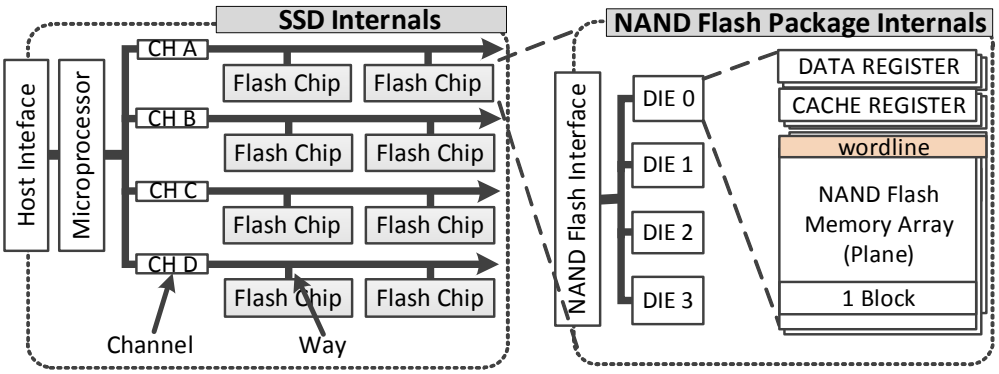


Figure 4.2: Typical SSD Architecture

and as high as 16 times for fully random synthetic I/O. For MLC drives this lifetime extension ranges from 73% to 375%, and reaches 325% in synthetic tests. Additionally, we provide time-series analysis demonstrating that ZombieNAND never degrades average latency compared to a vanilla drive during the lifetime of the vanilla drive. While average latency does, under some workloads, degrade somewhat after the death of the vanilla drive, a bit slower is much preferable to dead, and for around half of the traces latency is instead reduced by typically around 20% to 25%, and can reduce as much as 50% (even post-death) for TLC. MLC results demonstrate typical latency improvements from about 10% to 25% for half of the workloads, and can be as low as 40% reduced.

4.2 Background

To facilitate an informed vantage point for our work, we provide background starting at a high level for flash-based SSDs and drilling down. We first describe SSD architecture and some of the key parameters and mechanisms we explore in this work, and then move into the actual NAND flash itself. Having given a basic overview of the mechanisms in NAND flash, we last highlight some of the physics of NAND wear-out characteristics, which we leverage towards high-fidelity simulation of bit-switching for "undead" flash.

4.2.1 SSD Architecture

NAND-flash-based SSDs are highly parallel in nature, as shown in Figure 4.2. Each SSD is connected to the host typically by SATA, SAS, or PCIe interfaces, and multiple channels within connect the interface to groups of NAND packages. Within each package there exist one or more dies, and within each die there are often one or more planes, which require specialized data access modes to fully leverage. In total, this makes for an architecture where hundreds of individually small, low-bandwidth, flash storage areas all work in tandem to provide larger capacities, low latencies, and high bandwidth.

Managing that architecture towards optimal performance and importantly, effective wear-leveling, is a complex task that is the duty of the flash translation layer (FTL). This FTL performs duties including, but by no means limited to, address translation (between the host and the physical addresses), bad block management, wear-leveling, a variety of performance optimizations, and garbage collection. In this work the FTL plays a crucial role in allowing "undead" flash to not only clamber along after death, but also to acknowledge that these cells store a smaller capacity, and to utilize their special properties of being faster and longer-lived to extend the life and improve performance of the drive on the whole.

4.2.2 NAND Flash Overview

Moving down to analysis of the NAND flash material itself, at the level of an individual cell NAND is made up of floating-gate transistors. These transistors are capable of storing data for extended periods of time relative to DRAM and other temporary memory solutions, but have the side-effect that, in order to write over a previously written section, they must first be discharged (hereafter referred to as "erased"). For the most basic NAND flash cells, single-level cells (SLC), the floating-gate either has a high threshold voltage (V_t), which demarcates that the cell is erased and is a "1," or it has a low V_t, indicating it has been written and is "0." For increased bit counts, more voltage levels are added to indicate the increased number of bits, which divides the entire voltage range into 2^n distinct voltage levels for an n-bit flash cell.

4.2.3 The Physics of Cell Wear-Out

There are countless interesting and nuanced properties of NAND flash cells, but the critical ones for this work revolve around their physical properties relating to wearing-out. As mentioned, NAND flash suffers from wear-out due to the impact of the P/E cycle. This is specifically due to the charging process resulting in electrons getting "stuck" in the interface and oxide layers of the floating-gate transistor. As more and more electrons accumulate there, at some point V_t exceeds the margin between distinct voltage levels and therefore discerning between one level or the other becomes impossible. However, as is to be expected, this collection is not purely monotonic – after some time charge can "leak" out from the oxide, returning a cell from a dead state to a usable state again.

This process of electron trapping and leakage has been extensively studied and modelled in [66], [67], and [68], which we incorporate into this work. This level of high-fidelity wear-out simulation is required in this work because we need to know, if we switch from some higher bit-mode to a lower one, what "life" yet remains in the cells. Since switching is merely a logical interpretation, we retain the trapped voltage at a per-block granularity, which will not change on bit-switch, and reference a higher threshold relative to the current bit-mode.

Algorithm 4 Oxide Stress Model Psuedocode

```
1: procedure CALC_STRESS(cycle)
2:     A ← 0.08
3:     B ← 5.0
4:     Cox ← 2.15e^−17
5:     q ← 1.6e^−19
6:     δN_it ← A ∗ cycle^0.62
7:     δN_ot ← B ∗ cycle^0.30
8:     δV_it ← (δN_it ∗ q)/Cox
9:     δV_ot ← (δN_ot ∗ q)/Cox
10: return (δV_it + δV_ot)
11: end procedure
```

Pseudocode for our oxide stress model, most closely tracking the model as presented in [66], is shown in Figure 4. A and B in the model are constants derived in that work, and q is electron charge in Coulombs. As [66] failed to specify the value for Cox, which is the capacitance of the oxide, we were forced to re-calculate it by fitting to their graphs and validating against our evaluation on actual flash

as shown in Figure 4.1. δN_{it} and δN_{ot} represent densities of traps forming in the oxide, respectively, interface traps and bulk traps. Correspondingly, δV_{it} and δV_{ot} are the interface and bulk trap voltage shifts, which add together towards the total shift of δV_t.[1]

4.3 Wear-Unleveling for Lifetime

4.3.1 The Basics of Wear-Leveling

Modern SSDs tout all forms of complex garbage-collection and wear-leveling algorithms to assure the cells within wear-down at roughly the same rate throughout the many planes, dies, packages, and channels within. In doing this traditional SSDs, which simply mark cells as dead when they are no longer reliable in storing the bit-count they were designed for, do an excellent job at reaching near-optimal life. Optimal lifespan for a traditional SSD can be defined as "using all of the flash medium in such a uniform manner that when a cell finally is erased and declared dead, all of the remaining cells are also one erase from death themselves." In short, if all the flash dies at nearly the same time, the most use was gotten out of the pool of flash as is possible. To fully appreciate this consider a sub-optimal alternative, where half of the flash in the SSD is on the verge of death and the half has been completely unused thus far. If that near-dead half continues to be used and dies off, the drive will report itself as wholly dead when the amount of dead flash exceeds the reserved amount (often in the range of 5% to 30% extra flash). So, in this simplified case, only half of the drive's life has been expended even though it has reported itself as entirely dead to the operating system.

Wear-leveling functionality tends to be concentrated around the garbage-collection process as that is where erases largely occur, and erases have the most negative impact on cell lifetime. In a simple garbage collection process, once it begins on a given package it does not stop cleaning until a high-watermark is reached. At a high-level, first a usage table is retrieved, which organizes all the

[1] In private discussions with industry professionals we ascertained that, while this stress model roughly captures the effects of P/E cycles in modern NAND flash, recovery characteristics due to charge leakage from the oxide layers are far less consistent between manufacturers and feature sizes, and therefore difficult to predict. Therefore, to present the most conservative estimates of lifetime-enhancement, we assume zero recovery is possible.

blocks in the package into buckets based on the number of valid pages within. A page is considered valid if a logical address still points to it – upon an erase and/or rewrite of that logical address, the block is simply marked as "invalid," it is not necessarily immediately erased in physical terms. The highest priority bucket contains blocks whom have no valid pages within (and therefore no data will need to be read and written out somewhere else), and the lowest priority bucket has blocks with only one invalid page. Blocks without any invalid pages are not considered, as erasing and moving these does nothing but damage and slow the drive. By prioritizing erasure of blocks with large numbers of invalid pages, a smaller amount of data needs to be read out of these blocks, coalesced, and written back into other blocks (also known as write amplification).

This results in higher-performance garbage collection and lower write amplification, but without other protections does nothing to assure real wear-leveling is being achieved. Therefore, as the garbage collector works through the high-priority buckets, it also considers the number of times a given block has been erased in comparison to the average erase count of the entire package. If that block exceeds the average by a predefined threshold, we consider it to be overused and skip past it to another block with less erases. Again, this is a simplified garbage-collection and wear-leveling strategy – deciding which blocks to erase, which to skip past, how write-amplification should be balanced against over-worn blocks, and the like have all been studied at length in numerous other works. Moreover, with few exceptions, our proposed changes to this simplified wear-leveling mechanism as follows is sufficiently simple such that it should be orthogonally applicable to a broad base of the many garbage collectors and wear-leveling algorithms currently in the wild.

4.3.2 The Early Switching Pool

In this work a simplified wear-leveling strategy not only suffices to explore our methodology, but was found in early experiments to be overly effective. Put succinctly, when you give up the requirement that "a cell dies when it is no longer able to reliably store a predefined set of bits," and instead allow that cell to repurpose itself to a lower bit-count as we do, you no longer want your entire drive to reach a near-dead state at the same time. If such occurs, transitioning down to

a lower-bit count results in only a dead drive. If an MLC drive wholly transitions to SLC, unless the reserved area is greater than 50% of the drive (unlikely for cost reasons), the drive will be forced to report itself as dead. The same property applies to TLC, although the subsequent transition to SLC means an even greater reserved area would be needed to protect against whole drive transition without the drive announcing its death.

Therefore, we propose a simple but, as we will later demonstrate, highly effective adaptation to the above wear-leveling strategy that "wear-unlevels" a predefined fraction of the drive in a controlled manner *without sacrificing any of the manufacturer's guarantees about drive lifetime* as shown in Algorithms 5 and 6. Further, we define a pool of "early switch" blocks in each and every flash die as the following:

$$\text{Early Blocks} \leq \frac{(R - W) \times B}{2^{S-2}} \tag{4.1}$$

Here R is the reserved percentage, W is the high-watermark percentage (when GC stops), B is the number of blocks per element, and last S is the starting bit-level of the blocks (we do make the assumption that all blocks in the SSD start at the same bit-level). The high-water free blocks percentage is first removed from the pool of reserved blocks as we consider a drive dead if it exceeds this amount of dead blocks. The remaining percentage is then multiplied by the blocks per element to get our pool of blocks which can die without the element dying entirely, which we last divide by a power of two dependent upon the number of bits started with. This final term, which boils down to simply dividing by two if started as TLC, was decided upon after testing turning the entirety of that extra space into "early switch" cells. Because TLC (when compared against MLC) has a very low erase count prior to death, but a higher capacity after death and transition, getting a the early switch blocks to transition sooner is critical, and therefore concentrating accesses on a smaller early switch pool is the easiest way to achieve this.

Our wear-unleveling methodology as described in Algorithms 5 and 6 follow much of the same paths as the standard wear-leveling algorithm, with a two notable points to make: First, although unshown, whenever an erase is performed, if the block is discovered to be dead (in our simulator, indicated by having trapped charges exceeding specified thresholds), it is always transitioned down one bit (dead SLC can go no further and therefore is simply marked dead). Thresholds correspondingly raise on this action, and, provided the reserved capacity of the drive has not fallen

Algorithm 5 Usage Table Construction

```
1: procedure BUILD_USAGE_TABLE(element)
2:     table ← MALLOC_TABLE
3:     for i ← 0, params.blocks_per_element do
4:         usage ← metadata.block_usage[i].num_valid
5:         table[usage].len++
6:     end for
7:     buckets ← MALLOC_BUCKETS(table)
8:     for i ← 0, params.blocks_per_element do
9:         usage ← metadata.block_usage[i].num_valid
10:        buckets[usage][table[usage].tmp].num ← i
11:        buckets[usage][table[usage].tmp++].rem_life ←
       CALC_LIFE_REMAINING(element, i)
12:    end for
13:    for i ← 0, params.pages_per_block − 1 do
14:        QSORT(buckets[i], table[i].len, comp_life_earlyswitch)
15:        for j ← 0, table[i].len do
16:            table[i].block[j] ← buckets[i][j].num
17:        end for
18:    end for
19: end procedure
```

below the high-watermark percentage, the block can live on to die another day. Second, in creating our usage table buckets, number of valid pages is still the most important property considered in sorting. Simply being an early-switcher does not allow a block to get into a higher priority bucket – write amplification due to increased valid pages would over-ride potential benefits. However, if an early-switch block and a normal block are both in the same valid-pages bucket, the early-switch block will always be chosen, and thus the early-switch property is the second most important propery to sort upon. So, although early-switch blocks are not wear-leveled against normal blocks (and importantly, vice-versa), early-switch blocks are wear-leveled amongst themselves.

4.4 Evaluation

4.4.1 Simulation Framework

In order to evaluate the ZombieNAND algorithm as just described, we needed to first build a simulator capable of returning physically-accurate results for long-running

Algorithm 6 GC with Controlled Wear-Unleveling

```
1: procedure CLEAN_BLOCKS(element)
2:   avg_life ← COMPUTE_AVG_LIFE(element)
3:   usage_table ← BUILD_USAGE_TABLE(element)
4:   for i ← 0, params.pages_per_block − 1 do
5:     current_table ← usage_table[i]
6:     for j ← 0, current_table.length do
7:       blk ← current_table.block[j]
8:       rem_life ← CALC_LIFE_REMAINING(element, blk)
9:       if rem_life < (LIFE_THRESHOLD ∗ avg_life) then
10:        if NOT_EARLY_SWITCH(blk) then
11:          continue
12:        end if
13:      end if
14:      CLEAN_BLOCK(blk, element)
15:      if DONE_CLEANING(element) then
16:        break
17:      end if
18:    end for
19:    if DONE_CLEANING(element) then
20:      break
21:    end if
22:  end for
23: end procedure
```

simulations. DiskSim [69], a widely known and used magnetic disk simulator, has been extended for simulation of idealized SSDs by Microsoft Research [70]. As ZombieNAND concentrates on extending the overall lifetime of the SSD, we deemed it reasonably acceptable to use an idealized SSD simulation framework rather than something higher fidelity on the short-term but much more computationally expensive. However, this extension fell quite short of our needs as it only used block-counter style of lifetime estimation. Because we are switching the logical interpretation of a cell when it reaches death in its current state, it is not, for example, acceptable to simply deduct 3,000 P/E cycles that were used when in TLC state from the subsequent MLC state. This would result in 27,000 cycles remaining in MLC, and would significantly overestimate the remaining lifetime for the drives.

Therefore, after carefully reviewing works as discussed in Section 4.2, [66], [67], and [68], we incorporated their findings into a new, physically-aware lifetime

calculation sub-system within the DiskSim SSD Extension. This was the first and major hurdle in the road towards a simulator that would provide physically-accurate results for ZombieNAND. Second, we had to incorporate the notion of different page sizes in the simulator. Because DiskSim (and the extension) were built on the premise of single size sectors and pages, we had to carefully add features to the existing functionalities to enable them to cope with potentially two or three different page sizes (in the case of MLC or TLC, respectively). Last, although DiskSim is well known and used, it is also somewhat dated for modern 64-bit machines and to our knowledge has never been used for very long-running experiments. Specifically, to properly evaluate ZombieNAND, we needed to start with a fully pristine SSD, and continue issuing operations to it until it declared itself completely dead. This process would often take as long as a week on a single machine and would ultimately have issued *dozens of billions* of operations at the SSD over its lifetime. However, we found many of the variables and data structures in place failed to scale to handle these durations and raw volumes of commands. This resulted in a number of segmentation faults and less obvious bugs until we finally adapted all of the underlying code to handle long-running simulations like ours.

4.4.2 Experimental Setup

In evaluating ZombieNAND, we note the following configuration choices as shown in Tables 4.1 and 4.2. Five additional points are worth noting to fully understand the following experimental results: First, for the synthetic experiments we simulate an SSD with a single, reasonably small NAND flash element inside because we sought to explore a large search-space of parameters (which would be intractable at larger sizes and counts of elements). Second, in the case of the trace-driven experiments, we move to a larger, multi-element SSD sized at 1 Gigabyte, but cannot scale to modern sizes (128GB-1TB) because again, we are simulating the entire lifetime of the SSD. Even at 1GB, simulations take multiple days on modern machines, and DiskSim was not designed to scale up for HPC-style simulation (because of the large code-base, adapting it for such would be a massive undertaking). Nevertheless, we argue that our results should scale to larger sized SSDs for synthetic and trace-driven experiments so long as reserve area percentages, which we find to be the most dominant parameter, stay the same. Third, all of these experiments

Access Type (unit)	SLC (2KB)	MLC (4KB)	TLC (8KB)
Read (page)	0.025 ms	0.05 ms	0.15 ms
Write (page)	0.2 ms	0.5 ms	1.0 ms
Erase (block)	1.5 ms	1.5 ms	3.0 ms

Table 4.1: Access latencies based on operation type (unit operation works on is shown in parentheses) and NAND bit-level. NAND bit-level headers are shown with size of a page in parentheses.

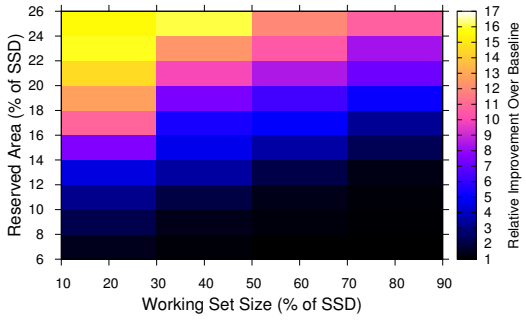
	Synthetic	Trace-Driven
Flash Chips	1	4
Blocks per Element	128	512
Planes per Element	8	8
Blocks per Plane	16	64
Pages per Block	128	128

Table 4.2: Key experimental configurations of the simulated SSD for synthetic and trace-driven tests.

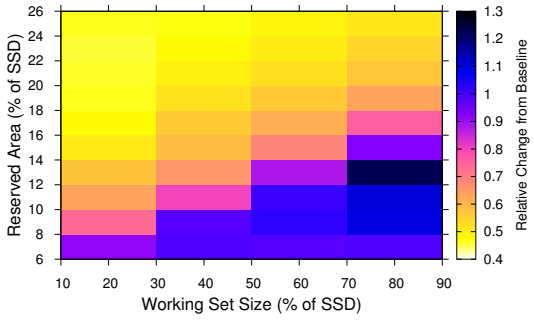
assume a SATA 300 interface (and corresponding block transfer times are used). Fourth, garbage collection, which has been explained to be a critical component of ZombieNAND, kicks in at a predefined minimum percentage of blocks free and runs in the background until a high-watermark percentage of blocks are available again. In all of our experiments, we use 2% and 5% for these values, respectively. Last, while we use a physics-based engine we developed based on models in prior works to determine lifetime remaining for a given flash cell, the parameters within have been tuned to correspond to state-of-the-art manufacturer guarantees. Specifically, we have tuned the physics engine to declare an *unswitching* SLC-, MLC-, and TLC-based SSD dead at roughly 75,000, 6,000, and 1,000 P/E cycles, respectively. These values were gathered from a broad survey we performed of recently-released SSDs in the various categories and manufacturer specifications associated with them, and the timings shown in Table 4.1 were drawn from specification sheets [71], [72], and [73].

4.4.3 Synthetic Results

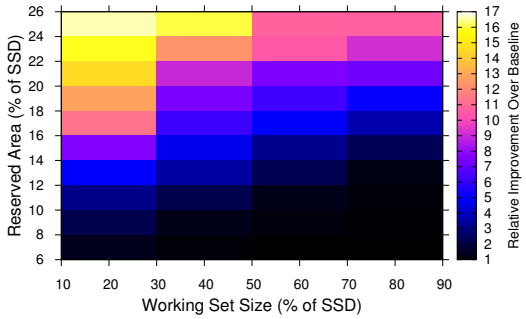
In this set of experiments we use a synthetic access generator we built that takes two parameters, read-to-write ratio and working-set size, and test it across a number of different reserved area configurations of the SSD. This synthetic access generator



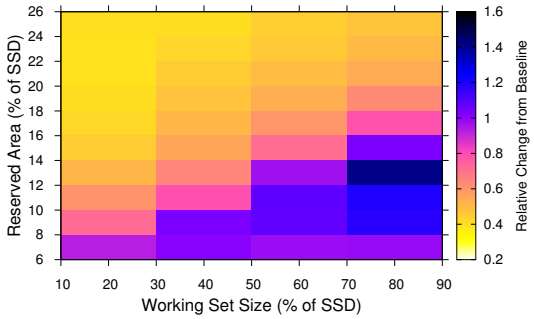
(a) Synthetic Lifetime (TLC 20% Writes)



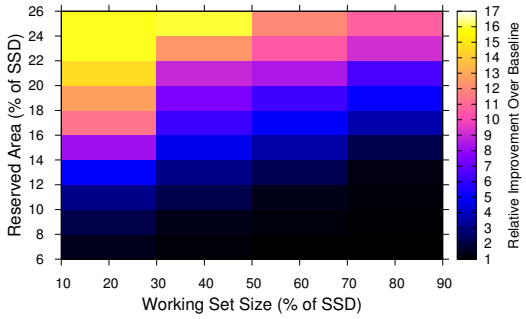
(b) Synthetic Latency (TLC 20% Writes)



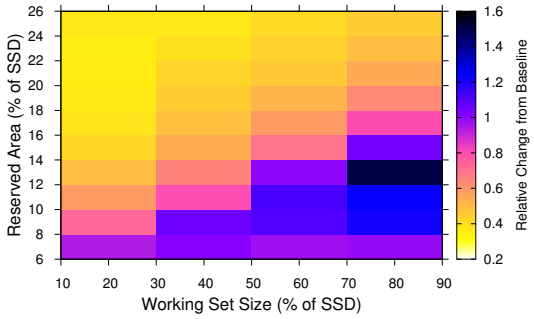
(c) Synthetic Lifetime (TLC 50% Writes)



(d) Synthetic Latency (TLC 50% Writes)

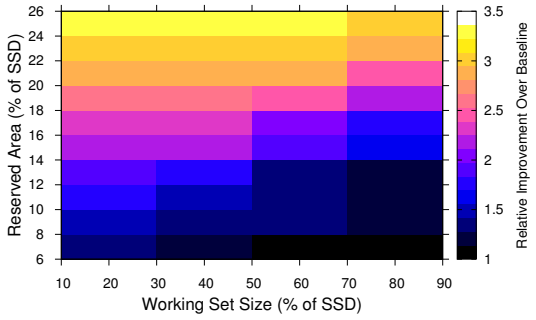


(e) Synthetic Lifetime (TLC 80% Writes)

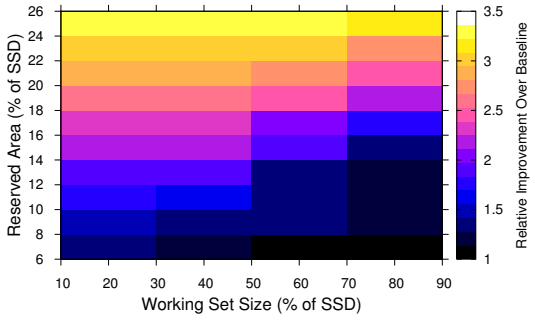


(f) Synthetic Latency (TLC 80% Writes)

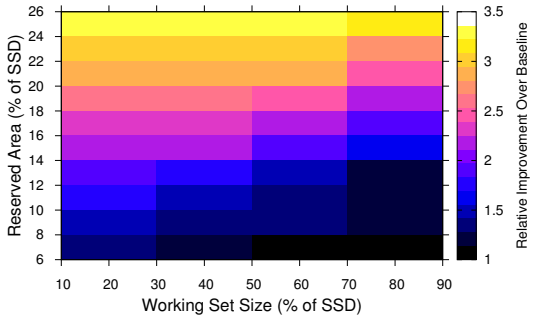
Figure 4.3: Lifetime and latency results for TLC-NAND-based SSDs under synthetically-generated random-I/O workloads. Varying amounts of reserved area, and working set size of the workload are shown within each graph, and varying amounts of writes are shown across graphs. Lighter colors indicate better lifetime or latency.



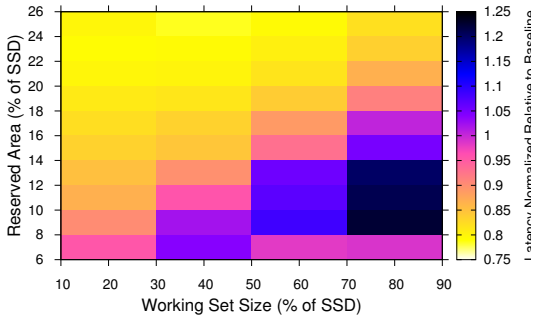
(a) Synthetic Lifetime (MLC 20% Writes)



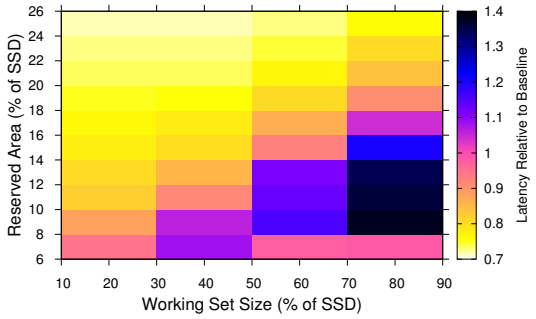
(b) Synthetic Lifetime (MLC 50% Writes)



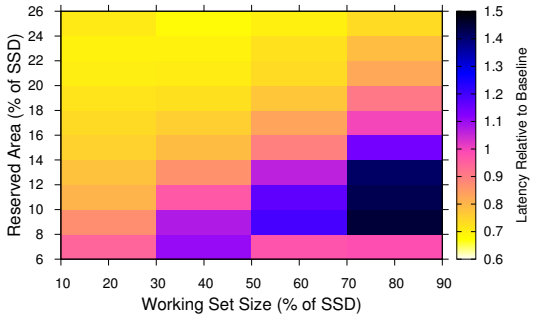
(c) Synthetic Lifetime (MLC 80% Writes)



(d) Synthetic Latency (MLC 20% Writes)



(e) Synthetic Latency (MLC 50% Writes)



(f) Synthetic Latency (MLC 80% Writes)

Figure 4.4: Lifetime and latency results for MLC-NAND-based SSDs under synthetically-generated random-I/O workloads. Varying amounts of reserved area, and working set size of the workload are shown within each graph, and varying amounts of writes are shown across graphs. Lighter colors indicate better lifetime or latency.

simply generates a random workload across the specified working-set size, and continues issuing random accesses as fast as the SSD can handle until the drive dies. **Read-to-write ratio** simply defines the probability the randomly generated access will be a read or a write. **Working-set size** defines how large of an address space the synthetic generator can randomly choose an access to occur on. For example, since we are using a single element in these experiments, 64MB in size, in the cases of 50% working set size, addresses starting at zero and going up to 32MB of the drive can all be issued to. Last, it is perhaps most important to understand specifically what we mean by **reserved area.** In all of our synthetic and trace-driven results, reserve area denotes the percentage of the total state size *deducted* from the user-visible SSD space. So, unlike its common use, in our 1GB SSD evaluation, whether we are using a reserved area of 10% or 30%, the total flash in the SSD is still 1GB – the only thing changing is the accessible logical address space to the application. This is a critical point because we did not want to confound our results by having added more reserved area to a base amount of flash; fixing the total raw flash available keeps all of the configurations on an even playing field.

Results from *over five-hundred distinct experiments* we performed to cover variations on these three key dimensions for both TLC and MLC are shown in Figures 4.3 and 4.4, respectively. These heat-maps depict life and latency changes *relative to an unswitching execution with the exact same parameters*, which we hereafter refer to as the *baseline*. The baseline is normalized to 1.0 for both latency and lifetime plots, and in the lifetime cases, all experiments do as least as well as the baseline (a design goal). The latency results depict *latency average over the entire lifetime of the SSD*, so it is important to remember in the ZombieNAND case the lifetime is always greater than or equal to the baseline. Therefore, even though sometimes the latencies are worse than the baseline average, *these degradations only occur after the life of the baseline SSD has expired*. We lack space to show detailed, time-series latency results for these synthetic runs, but for an example of this behavior, please reference Figure 4.7 from the results of trace-driven evaluation.

From these synthetic results, we identify the following major three take-aways:

First, for TLC drives, small working-set sizes and large reserve areas result in enormous gains over the baseline in excess of an order of magnitude. At first blush these gains seem egregious, but bear in mind we are starting at TLC, which has a

lifetime around 1,000 P/E cycles, and we transition to SLC, which has a lifetime 75 times greater. Obviously a full 75 times improvement is not expected as that is not how the physics of flash lifetime works, nor can the entire drive be converted to SLC and still live, but these (still large) improvements are a function of the great difference in endurance between the types. The MLC results depict a similar inter-relationship as we vary the parameters, however demonstrate lower gains possible than in the TLC case. Again, this relates to the difference in lifetimes – as MLC lasts approximately six times longer than TLC from the outset, transitioning to SLC buys us some, but not as much relative lifetime as in the case of starting at TLC. Similarly, latency differences between MLC and SLC are smaller than TLC to SLC, so these gaps (both in lifetime improvements and degradations) also lessen.

Second, we are witness to a somewhat odd latency degradation in the TLC case not in the bottom right as we would expect (highest working size, lowest reserved area) but somewhat up from that. In analyzing the results, we find that for reserved areas smaller than around 13%, there is no real change in lifetime or latency. Right around 13% lifetime begins to extend, but, because there is limited amounts of converted "fast blocks," all too often an access results in a write to a fast block *and* a normal block (because the access is larger than the smaller, fast blocks size, and no other fast blocks may be available at that time). This results in latencies somewhat larger than the normal block access until the reserve area grows enough that a decent pool of fast blocks is available at any given time in most of the invalid-page buckets.

Third, and what may have been initially most apparent, we witness extremely similar *relative* behavior across low, medium, and high read-to-write ratio graphs. We emphasize relative because the absolute number of accesses is definitely not the same (since reads have extremely limited impact on cell lifetime) across equivalent configurations where only read-to-write ratio is changed.

4.4.4 Trace-Driven Results

Last, we consider the efficacy of ZombieNAND in extending SSD lifetime and explore its impact on latency over a variety of real application traces. These ten traces represent I/O workloads in financial workloads, file servers, user home directories, and internet SQL servers, derived from traces available at [74] and [75]. We have

Application Trace	Writes (in %)			Reads (in %)		
	Total	Random	Sequential	Total	Random	Sequential
Fin-A	83	47	53	17	45	55
Fin-B	22	33	67	78	78	22
NFS-A	99	1	99	1	1	99
NFS-B	58	22	78	42	1	99
NFS-C	19	0	100	81	0	100
User-A	27	23	77	73	8	92
User-B	9	45	55	91	2	98
User-C	58	18	82	42	6	94
SQL-A	43	25	75	57	10	90
SQL-B	15	26	74	85	2	98

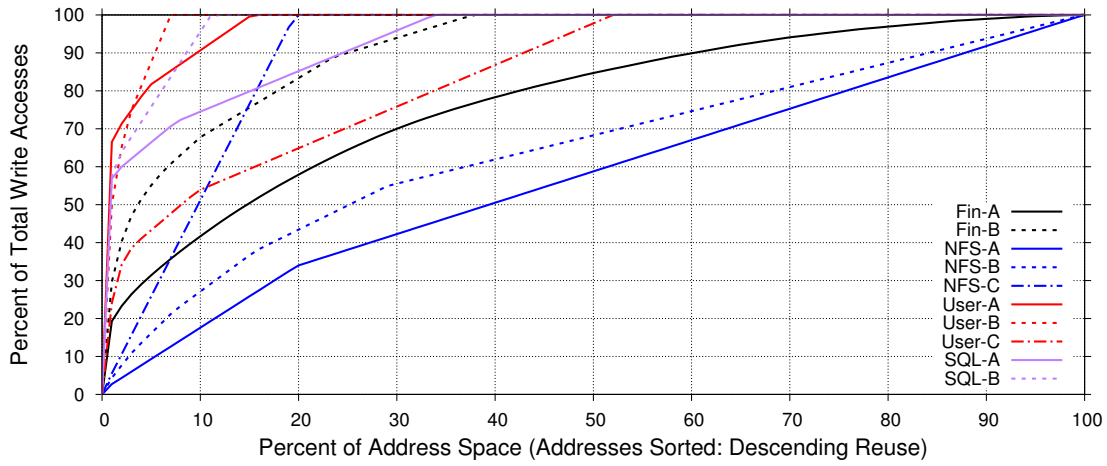
Table 4.3: Application trace access composition. Percents expressed in terms of raw data moved.

quantified and present specifics about each trace, including the read-to-write ratio, random vs. sequentiality of the trace, and address reuse of accesses in Table 4.3.

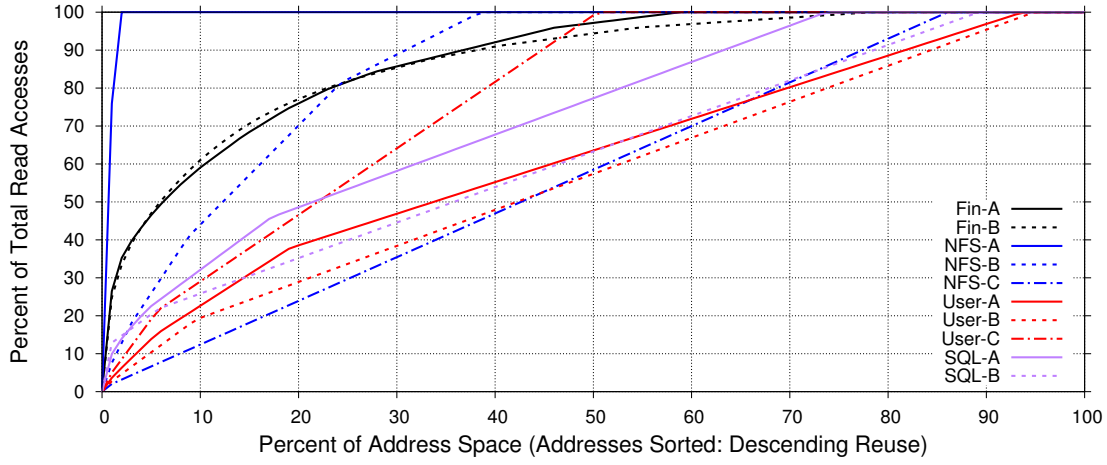
In order to keep our results from becoming entangled amidst too many varying factors, we only use the first 512MB of addresses from every trace. This means some traces run very close to only issuing 512MB of accesses (very low address reuse), whereas others issue many gigabytes of accesses (high address reuse) before hitting the 512MB of addresses limit. We must do this to a) make sure the traces do not attempt to use more space than is available in our 1GB SSD, and b) to normalize the address space accessed amongst all traces to roughly 50% of the drive.

This does not in any way mean address reuse has been normalized across traces, as can be seen in Figure 4.5. Those graphs sort by descending reuse and provide volumes of accesses on the y-axis, demonstrating that while some traces exhibit largely uniform accesses across a majority of the access space (e.g., NFS-C in Figure 4.5b), others only use a small fraction of the entire space for all of their accesses (e.g., Fin-B in Figure 4.5a). We provide this level of detail because address reuse is a driving factor in the efficacy of ZombieNAND, as we will explain in a moment.

We begin by dissecting the lifetime improvements brought by employing ZombieNAND as shown in the overlaid bar graph in Figure 4.6. All lifetimes shown are relative to the baseline, which has been normalized to 1, and all perform as good or better than the baseline. Further, increased reserve areas never results in degraded results, so all results are overlaid to demonstrate how much life an additional 10%



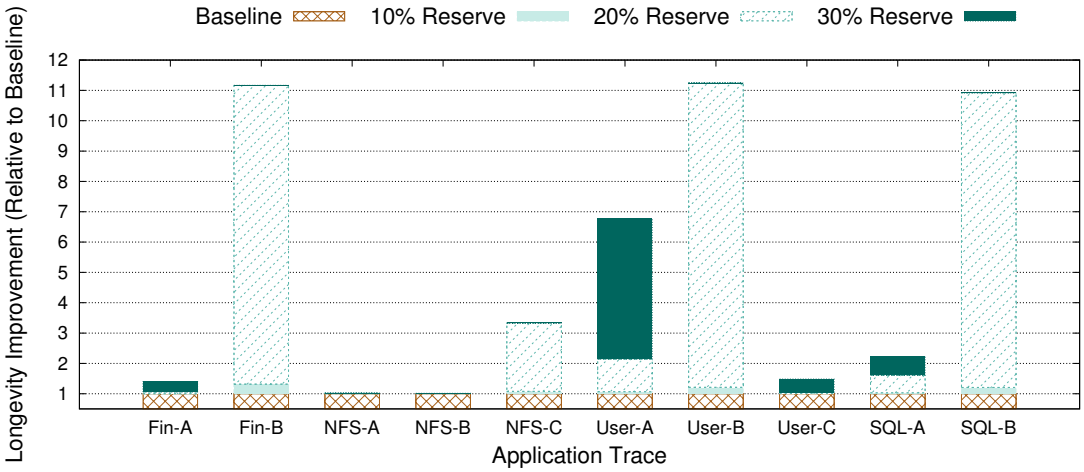
(a) Trace Address Reuse (Writes)



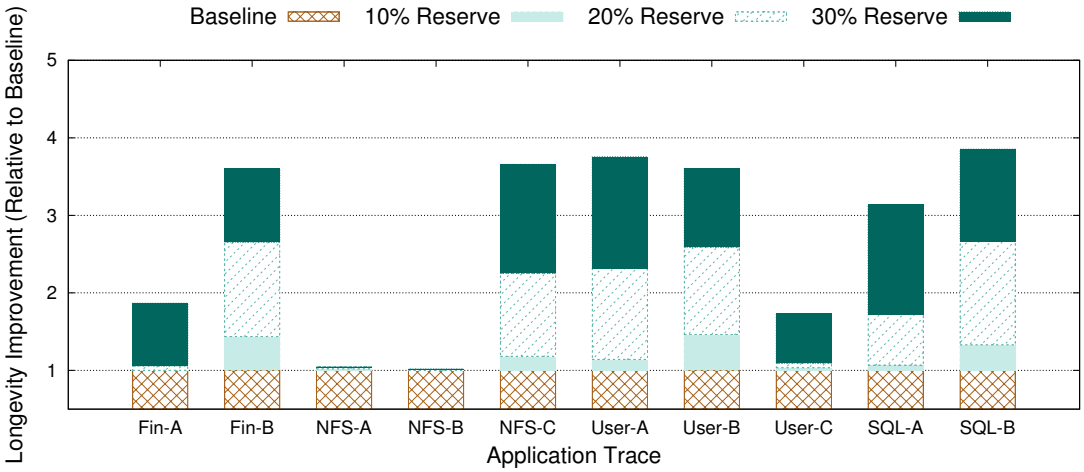
(b) Trace Address Reuse (Reads)

Figure 4.5: CDF plots, which describe the fraction of total accesses commonly used addresses are accountable for. For instance, steep curves indicate a small fraction of the address space of a specific trace accounts for a large percentage of the total accesses, and low, slowly growing curves indicate fairly balanced accesses to a large portion of the address space.

of reserved area buys us. For instance, in TLC SSD executions of User-A, we can see that while 10% reserve provides extremely small improvements over the baseline, expanding to use of 20% reserve achieves a factor of 2, and moving to 30% gets us just short of a factor of 7. Interestingly, this trend is not consistent amongst the traces – some benefit from 10%, but skyrocket at 20% and do not improve further at 30%, while others do not benefit from 10% but equally improve from 20% and 30%, and yet others do not benefit from any amount of reserve.



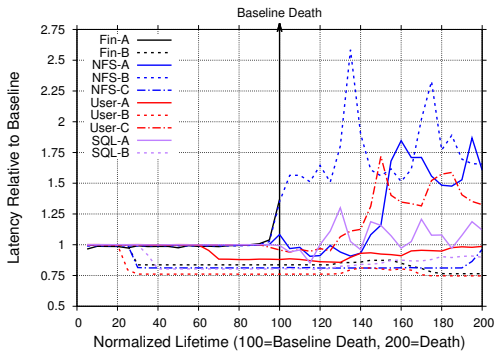
(a) Trace Lifetime Comparison (TLC)



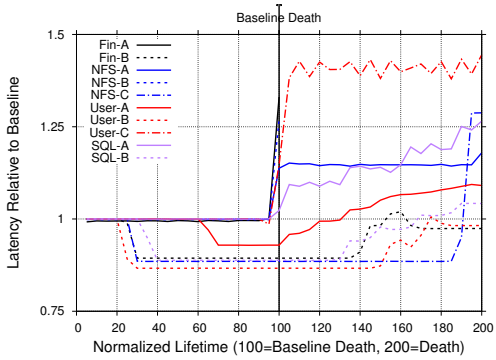
(b) Trace Lifetime Comparison (MLC)

Figure 4.6: Length of lifetimes shown relative to baseline (normalized to 1.0) for varying reserve areas. Our algorithm always does at least as good as the baseline, and increased reserves always do as good or better than a smaller one. As shown, some applications benefit tremendously (an order of magnitude in 3 of the 10 traces) from slight increases in reserves, but may not benefit from increased reserves beyond that, whereas others demonstrate little or no improvement over the baseline.

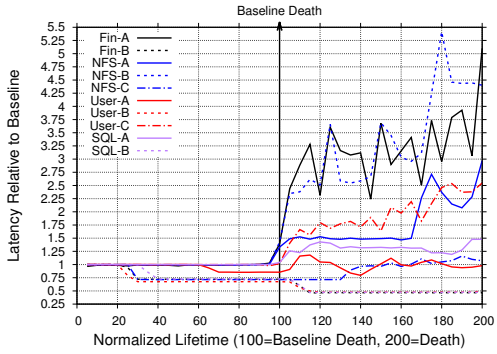
Analyzing these findings carefully next to the trace breakdown table and CDF plots as presented earlier, we identify one clear and defining driver for ZombieNAND improvements: In order for ZombieNAND to have real impact, writes must be concentrated into a reasonably small fraction of the total address space. Looking to our previous synthetic results, this trend is echoed there, where the working-set size more or less defines performance. What is starkly different from those synthetic



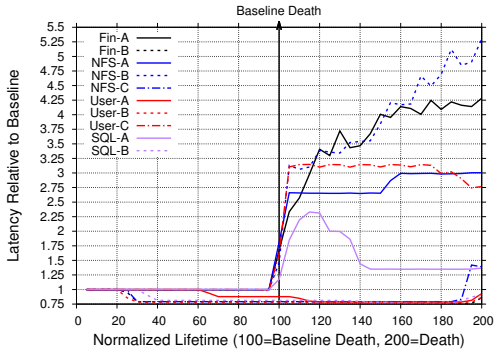
(a) Latency Change over Life (TLC 10% Reserved)



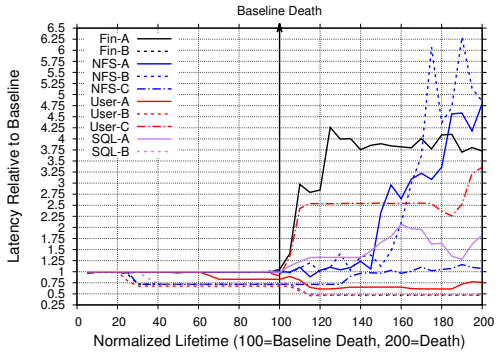
(b) Latency Change over Life (MLC 10% Reserved)



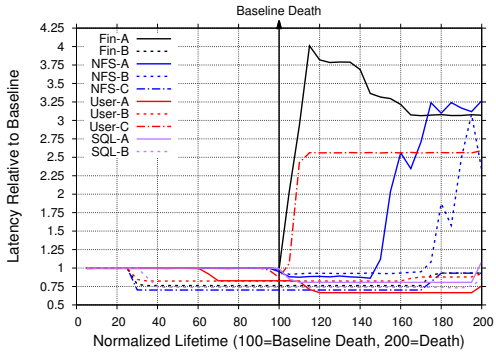
(c) Latency Change over Life (TLC 20% Reserved)



(d) Latency Change over Life (MLC 20% Reserved)



(e) Latency Change over Life (TLC 30% Reserved)



(f) Latency Change over Life (MLC 30% Reserved)

Figure 4.7: Latencies over the entire lifetimes of the traces explored, broken down into 10, 20, and 30 percent reserve areas for both MLC- and TLC-based SSDs and normalized over a 200 point range. Vertical line at x-axis 100 demarcates death of the baseline, and, if our algorithm improves lifetime for that trace, x-axis 200 indicates death of the ZombieNAND-enhanced drive. Latencies all shown relative to the baseline, which has been normalized to 1.0.

results is that such randomized uniformity is not presented in any of these traces, and therefore increasing the reserves does not always net improvements. In fact, in our best performers, going from 20% to 30% reserve nets nothing additional. Analysis of the address reuse graph provides evidence suggesting why: Unlike randomized, uniform accesses as in the synthetic graphs, Fin-B, User-B, and SQL-B all reach different volumes of accesses at different percentages of commonly used addresses (a CDF of the synthetic I/Os would result in linear lines with varying slopes depending upon working-set size). The high-performers tend to have very aggressive slopes in the first few percent of the most commonly used address space, and reach very near to 100% of all accesses performed prior to 30-40% of the address space. Further, also unlike the simplified environment of the synthetic tests, read-heavy workloads do demonstrate more significant lifetime improvements than their mixed brethren with a similar write address reuse curve.

Moving onto the MLC trace-driven results, we are witness to expectedly lower relative improvements, but more consistency in terms lifetime enhancement across the traces. In fact, with the exception of NFS-A and NFS-B, which still do not improve hardly at all (due to lack of write address reuse), all of the traces gain a significant amount of lifetime relative to the best case. Furthermore, the traces also more frequently take advantage of both the 20% and the 30% jump in reserve area, unlike TLC, and likely due to the relative space differences between MLC and TLC.

Finally, the time-series analysis of latency in Figure 4.7 sheds considerably more light on the latency situation for ZombieNAND than the average latency values did earlier. While we do not have space to fully explore the rich graphs there, two important take-aways exist: One, although ZombieNAND does a great job of leveraging improved latencies for deceased blocks early on in the life of the drive for those traces it can benefit (prior to baseline death), it does not hurt the traces which do not benefit as much from the scheme until the baseline is within 5% from death. Two, although it is not a rule, while increased reserve does tend to improve lifetime, it also seems to have a negative impact on the post-baseline-death latencies for some of the traces. This is likely due to increased contention for the zombified block pool and "spillover" accesses that stretch across zombified blocks and normal blocks.

4.5 Related Work

We divide related works into three categories: First, those which laid the hardware and physics foundation to perform this switching. Second, those which employ bit-switching preemptively for performance reasons. And third, those which employ bit switching for the same reasons as we do, but to lesser or different effect.

First, in [76] Taehee Cho et al. propose a dual-mode flash memory technology, which offers both SLC-mode operations and MLC-mode operations in the same NAND flash fabrication. While MLC mode operations use a low-voltage-based incremental step pulse programming (ISPP) method to tightly control the cell thresholds, SLC-mode employs three times higher voltage to reach high throughput. Though this dual-mode NAND flash demonstrates that multiple modes are possible on the same flash die device, they do not tackle latency variation, reliability issues, or side-effects resulting from this architecture.

Moving onto those works which use bit-switching preemptively for performance reasons, we see in [77] Moinuddin K. Qureshi et al. propose a Morphable Memory System (MMS) that utilizes different latency values as observed in MLC phase change random access memory (PCRAM). Based on incoming request and workloads, MMS uses different resistance values on PCRAM in an attempt to reduce memory operational latency. Even though MMS provides insights regarding bit-switching, it does not deal with reliability whatsoever, and furthermore the access granularities and resistance characteristics explored differ markedly from those in NAND flash making MMS inapplicable to this work. In [78], Sungjin Lee et al. propose a flexible flash file system (FlexFS), which partitions NAND flash into two sections of SLC and MLC, dynamically sizing each based on applications requirements. This mode switching is used to powerful effect to satisfy quality of service requirements, however, similar to MMS, FlexFS also ignores reliability issues. Last in this group, in [79], Laura M. Grupp, et. al., propose a flexible flash translation layer, which mimics FlexFS in a number of ways. Specifically, it schedules performance-critical operations and bursty workloads using SLC, and achieves such in a TLC/MLC/SLC device by revealing latency variation patterns for both SLC, MLC and TLC amongst pages. Yet again, this only handles performance characteristics, and importantly, all three of these works preemptively adapt the capacity of the drive, which renders it incompatible with traditional file systems

and unaligned with manufacturer reliability guarantees.

Last, looking at a selection of works which most aligns with ours, we find three in particular that take a similar approach but either do so only superficially or do so for a different NVM type. In [80], Xavier Jimenez et. al. present, in their interactive poster, a topical exploration of how reviving dead blocks might impact lifetime. However, this work fails to capture all of the benefits of our approach for a number of reasons: First and foremost, they demonstrate very marginal improvements because they fail to fully explore the parameter space of modern SSDs, including, but not limited to, not looking at varying reserved areas, varying working set sizes, and a wide range of applications as we do in this work. Second, they do not examine the impact of adapting the wear-leveling schemes in modern SSDs to improve the longevity and performance of SSDs as they age. Last, rather than exploring this for a general SSD architecture, they only explore a narrow use-case: Hybrid FTL architectures. By bringing their dead MLC blocks back as SLC just for buffering for their log-structured FTL, this results in the healthier MLC still being written to when the logs are written out, resulting in write amplification compared to our use case and reads failing to leverage the vastly improved speeds in SLC. Moving onto [81], Azevedo et. al. propose "Zombie Memory," which on its surface sounds very similar to our work. However, their work actually explores an orthogonal approach to lifetime extension, and one which delivers no performance benefits for their MLC PCM. In fact, as their PCM ages and their techniques come to the fore, the performance degrades. Specifically, they look at using error correcting codes across partially dead blocks to extend one block's lifetime by sourcing parts of other dead blocks, and they never consider changing the bit mode from MLC to SLC; whatever bit-mode it starts as, it stays that way. This scheme could be applied in tandem with ours to multiplicative lifetime gains, a consideration of ours for future work. Last, in [82], Dong et. al. propose AdaMS, an adaptive PCM design to switch from MLC to SLC when PCM-specific failure events occur, and they design circuitry around this specific functionality to cope with the transition. Because the failure events, lifetime model used, circuitry designed, and algorithm proposed all rely on very specific PCM characteristics, which NAND flash do not share, their approach, while similar, is not applicable to NAND flash storage.

4.6 Conclusion

As widespread adoption and new uses arise for NAND-flash-based SSDs, pressure for higher density will only increase. To achieve this, manufacturers very well may continue stuffing more bits into flash cells, or continue shrinking them further when they are already struggling to retain data at the current feature size generation. Without intervention, this will reignite worries over flash longevity people were finally beginning to get over. Enter ZombieNAND, our novel NAND-cell-reviving scheme with accompanying modified garbage-collection and "wear-unleveling" algorithm to do something with those otherwise deceased and unusable flash cells, and which does so without jeopardizing any manufacturer-specified guarantees of P/E cycles.

Using a heavily-modified simulator we added a flash physics engine to so we could predict lifetime of changing cells with high-fidelity, we evaluate and analyze ZombieNAND across over five-hundred synthetic and sixty application trace-driven experiments. We demonstrate that ZombieNAND succeeds in extending the lifetime of TLC and MLC SSDs, sometimes in excess of an order of a magnitude, and for most of the traces and synthetic configurations run over two times. Moreover, we show that it absolutely does not deteriorate the lifetime for workloads that it cannot help, and equally importantly, does not degrade the latency of runs until after a normal SSD without ZombieNAND on-board would already be dead. In fact, for around half of our experiments, it consistently delivers in excess of 25% faster latencies than the baseline. Finally, we provide thorough analysis of these results and detail the property of the traces used to gather them.

Chapter 5
Conclusion

In a computing landscape increasingly "data-driven," this dissertation highlights the existing and ever-growing irony that simply getting to said data becomes a more costly, lengthy, and complicated process by the day. Driven in large part by a widening bandwidth gap between the processor and stable storage, herein we have proposed and explored a protean approach to shortening or constructing bridges across that gap in three major ways:

First, in RainFS, we enable data consolidation in the face of rampant storage fragmentation, which serves to bring as many storage units into the same pool as possible, reduce data duplication, and maximize aggregate performance. Second, we develop and demonstrate that *intelligent* data reduction is possible with the advent of tools like TreeChunks, which quantify the impact both in terms of performance and capacity change. By enabling intelligent data reduction evaluation, old and new schemes alike can be more readily adopted to fit more data over an equally sized bridge across the bandwidth gap. Third and last, we explore a new scheme ZombieNAND to overcome the longevity hurdle NAND-flash-based SSDs face in displacing their slower, spinning, magnetic brethren. Considering these three approaches in tandem, we argue future data storage scientists are far better equipped to cope with the increasing bandwidth gap at their doorstep.

Bibliography

[1] Box, G. E. and N. R. Draper (1987) *Empirical model-building and response surfaces.*, John Wiley & Sons.

[2] Anderson, C. (2008) "The end of theory," *Wired magazine*, **16**.

[3] Norvig, P., "All we want are the facts, ma'am," `http://norvig.com/fact-check.html`.

[4] "TOP500 Supercomputer Sites," `http://www.top500.org/`.

[5] Petitet, A., R. C. Whaley, J. Dongarra, and A. Cleary (2008), "HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers," `http://www.netlib.org/benchmark/hpl/`.

[6] Schaller, R. R. (1997) "Moore's law: past, present and future," *Spectrum, IEEE*, **34**(6), pp. 52–59.

[7] Shiroishi, Y., K. Fukuda, I. Tagawa, H. Iwasaki, S. Takenoiri, H. Tanaka, H. Mutoh, and N. Yoshikawa (2009) "Future Options for HDD Storage," *Magnetics, IEEE Transactions on*, **45**(10), pp. 3816–3822.

[8] Kryder, M. H., E. C. Gage, T. W. McDaniel, W. A. Challener, R. E. Rottmayer, G. Ju, Y.-T. Hsia, and M. F. Erden (2008) "Heat assisted magnetic recording," *Proceedings of the IEEE*, **96**(11), pp. 1810–1835.

[9] Richter, H., A. Dobin, O. Heinonen, K. Gao, R. Veerdonk, R. Lynch, J. Xue, D. Weller, P. Asselin, M. Erden, et al. (2006) "Recording on bit-patterned media at densities of 1 Tb/in and beyond," *Magnetics, IEEE Transactions on*, **42**(10), pp. 2255–2260.

[10] Wood, R., M. Williams, A. Kavcic, and J. Miles (2009) "The feasibility of magnetic recording at 10 terabits per square inch on conventional media," *Magnetics, IEEE Transactions on*, **45**(2), pp. 917–923.

[11] Dean, J. and S. Ghemawat (2008) "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, **51**(1), pp. 107–113.
URL `http://doi.acm.org/10.1145/1327452.1327492`

[12] Ghemawat, S., H. Gobioff, and S.-T. Leung (2003) "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM, New York, NY, USA, pp. 29–43.
URL http://doi.acm.org/10.1145/945445.945450

[13] Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber (2008) "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, **26**(2), pp. 4:1–4:26.
URL http://doi.acm.org/10.1145/1365815.1365816

[14] "Hadoop," http://hadoop.apache.org/.

[15] Borthakur, D., "The Hadoop Distributed File System: Architecture and Design," hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.

[16] Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber (2006) "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association, Berkeley, CA, USA, pp. 205–218.
URL http://dl.acm.org/citation.cfm?id=1298455.1298475

[17] Hindman, B., A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica (2011) "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, USENIX Association, Berkeley, CA, USA, pp. 22–22.
URL http://dl.acm.org/citation.cfm?id=1972457.1972488

[18] "Apache Hadoop NextGen MapReduce (YARN)," http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[19] et. al., W. G. (2009) *MPI: A Message-Passing Interface Standard*.

[20] Ligon, . I., W. B. and R. B. Ross (1996) "Implementation and Performance of a Parallel File System for High Performance Distributed Applications," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, IEEE Computer Society, Washington, DC, USA, pp. 471–.
URL http://dl.acm.org/citation.cfm?id=525592.823101

[21] "The Lustre File System," http://wiki.lustre.org/index.php/Main_Page.

[22] Schmuck, F. and R. Haskin (2002) "GPFS: A Shared-disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, USENIX Association, Berkeley, CA, USA, pp. 16–16.
URL http://dl.acm.org/citation.cfm?id=1973333.1973349

[23] Welch, B., M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou (2008) "Scalable Performance of the Panasas Parallel File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USENIX Association, Berkeley, CA, USA, pp. 2:1–2:17.
URL http://dl.acm.org/citation.cfm?id=1364813.1364815

[24] Buck, J. B., N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt (2011) "SciHadoop: Array-based Query Processing in Hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, New York, NY, USA, pp. 66:1–66:11.
URL http://doi.acm.org/10.1145/2063384.2063473

[25] Buck, J., N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, N. Polyzotis, and A. Torres (2013) "SIDR: Structure-aware Intelligent Data Routing in Hadoop," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, ACM, New York, NY, USA, pp. 73:1–73:12.
URL http://doi.acm.org/10.1145/2503210.2503241

[26] et. al., S. S. (2003), "Network File System (NFS) version 4 Protocol," .

[27] Hertel, C. (2003) *Implementing CIFS: The Common Internet File System*, Prentice Hall Professional Technical Reference.

[28] "Amazon S3," http://aws.amazon.com/s3/.

[29] "KosmosFS," http://code.google.com/p/kosmosfs/.

[30] Tantisiriroj, W., S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross (2011) "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, New York, NY, USA, pp. 67:1–67:12.
URL http://doi.acm.org/10.1145/2063384.2063474

[31] "PoweredBy Hadoop," http://wiki.apache.org/hadoop/PoweredBy.

[32] O'Malley, O. (2008) "Terabyte sort on apache hadoop," Yahoo.

[33] O'Malley, O. and A. Murthy (2009) "Winning a 60 second dash with a yellow elephant," Proceedings of sort benchmark.

[34] "Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster," http://www.top500.org/system/177999.

[35] Mihailescu, M., G. Soundararajan, and C. Amza (2013) "MixApart: Decoupled Analytics for Shared Storage Systems," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, USENIX Association, Berkeley, CA, USA, pp. 133–146.
URL http://dl.acm.org/citation.cfm?id=2591272.2591287

[36] Ananthanarayanan, R., K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari (2009) "Cloud Analytics: Do We Really Need to Reinvent the Storage Stack?" in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, USENIX Association, Berkeley, CA, USA.
URL http://dl.acm.org/citation.cfm?id=1855533.1855548

[37] "EMC Isilon OneFS," http://www.emc.com/domains/isilon/index.htm.

[38] "The NetApp OpenSolution for Hadoop," http://www.netapp.com/us/solutions/big-data/hadoop.aspx.

[39] Inc., P., ".ZIP File Format Specification," .
URL http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[40] loup Gailly, J., "gzip," .
URL http://www.gnu.org/software/gzip/gzip.html

[41] Seward, J., "bzip2," .
URL http://www.bzip.org/

[42] Corporation, M., "Microsoft Cabinet Format," .
URL http://msdn.microsoft.com/en-us/library/bb267310.aspx

[43] Pavlov, I., "7-Zip," .
URL http://www.7-zip.org/

[44] Roshal, E., "WinRAR Archiver, a powerful tool to process RAR and ZIP files," .
URL http://www.rarlab.com/

[45] N/A, "BTRFS Wiki," .
URL https://btrfs.wiki.kernel.org/index.php/Main_Page

[46] Corporation, O., "ZFS," .
URL http://www.opensolaris.org/os/community/zfs/

[47] Corporation, M., "NTFS Technical Reference," .
URL http://technet.microsoft.com/en-us/library/cc758691%28WS.10%29.aspx

[48] Ruijter, M., "lessfs: Open Source data de-duplication," .
URL http://www.lessfs.com/wordpress/

[49] Corporation, N., "NetApp Deduplication and Compression," .
URL http://www.netapp.com/us/products/platform-os/dedupe.aspx

[50] Corporation, E., "Data Domain Deduplication Storage Systems," .
URL http://www.emc.com/data-protection/data-domain/data-domain-deduplication-storage-systems.htm

[51] Corporation, M., "Data Deduplication Overview," .
URL http://technet.microsoft.com/en-us/library/hh831602.aspx

[52] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3," .
URL https://tools.ietf.org/html/rfc1951

[53] Huffman, D. A. et al. (1952) "A method for the construction of minimum redundancy codes," *Proceedings of the IRE*, **40**(9), pp. 1098–1101.

[54] Team, C., "The LZ77 algorithm," .
URL http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/lz77.html

[55] Blelloch, G. E. (2001), "Introduction to data compression," .
URL http://www.cs.cmu.edu/~guyb/realworld/compression.pdf

[56] N/A, "LZ4 Extremely Fast Compression Algorithm," .
URL http://code.google.com/p/lz4/

[57] ———, "BTRFS Wiki - What happens to incompressible files?" .
URL https://btrfs.wiki.kernel.org/index.php/Compression#What_happens_to_incompressible_files.3F

[58] Bolosky, W. J., S. Corbin, D. Goebel, and J. R. Douceur (2000) "Single Instance Storage in Windows® 2000," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, USENIX Association, Berkeley, CA, USA, pp. 2–2.
URL http://dl.acm.org/citation.cfm?id=1267102.1267104

[59] Rivest, R. (1992), "The MD5 message-digest algorithm," .
URL http://tools.ietf.org/html/rfc1321

[60] Eastlake, D. and P. Jones (2001), "US secure hash algorithm 1 (SHA1)," .

[61] of Standards, N. I. and Technology, "Descriptions of SHA-256, SHA-384, and SHA-512," .
URL http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

[62] Quinlan, S. and S. Dorward (2002) "Venti: A New Approach to Archival Storage," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, USENIX Association, Berkeley, CA, USA, pp. 7–7.
URL http://dl.acm.org/citation.cfm?id=1973333.1973340

[63] Rabin, M. O. (1981) *Fingerprinting by random polynomials*, Center for Research in Computing Techn., Aiken Computation Laboratory, Univ.

[64] Soundararajan, G., V. Prabhakaran, M. Balakrishnan, and T. Wobber (2010) "Extending SSD Lifetimes with Disk-based Write Caches," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, USENIX Association, Berkeley, CA, USA, pp. 8–8.
URL http://dl.acm.org/citation.cfm?id=1855511.1855519

[65] Cai, Y., E. F. Haratsch, O. Mutlu, and K. Mai (2013) "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp. 1285–1290.
URL http://dl.acm.org/citation.cfm?id=2485288.2485597

[66] Mohan, V., T. Siddiqua, S. Gurumurthi, and M. R. Stan (2010) "How I Learned to Stop Worrying and Love Flash Endurance," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, USENIX Association, Berkeley, CA, USA, pp. 3–3.
URL http://dl.acm.org/citation.cfm?id=1863122.1863125

[67] Lee, S., T. Kim, K. Kim, and J. Kim (2012) "Lifetime Management of Flash-based SSDs Using Recovery-aware Dynamic Throttling," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, USENIX Association, Berkeley, CA, USA, pp. 26–26.
URL http://dl.acm.org/citation.cfm?id=2208461.2208487

[68] Godard, B., J.-M. Daga, L. Torres, and G. Sassatelli (2007) "Evaluation of Design for Reliability Techniques in Embedded Flash Memories,"

in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, EDA Consortium, San Jose, CA, USA, pp. 1593–1598.
URL http://dl.acm.org/citation.cfm?id=1266366.1266716

[69] Bucy, J. S., J. Schindler, S. W. Schlosser, and G. R. Ganger (2008) "The DiskSim Simulation Environment Version 4.0 Reference Manual," .

[70] Agrawal, N., V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy (2008) "Design Tradeoffs for SSD Performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, USENIX Association, Berkeley, CA, USA, pp. 57–70.
URL http://dl.acm.org/citation.cfm?id=1404014.1404019

[71] *NAND Flash Memory MT29F32G08ABAAA, MT29F64G08AFAAA SLC datasheet*, *Tech. rep.*, Micron Technology, Inc.

[72] *NAND Flash Memory MT29F8G08MAAWC, MT29F16G08QASWC MLC datasheet*, *Tech. rep.*, Micron Technology, Inc.

[73] *NAND Flash Memory MT29F64G08EBAA TLC datasheet*, *Tech. rep.*, Micron Technology, Inc.

[74] Bates, K. and B. McNutt, "UMASS Trace Repository," .
URL traces.cs.umass.edu/index.php/Main/Traces

[75] "SNIA IOTTA Repository," .
URL http://iotta.snia.org/tracetypes/3

[76] Cho, T., Y.-T. Lee, E.-C. Kim, J.-W. Lee, S. Choi, S. Lee, D.-H. Kim, W.-G. Han, Y.-H. Lim, J.-D. Lee, J.-D. Choi, and K.-D. Suh (2001) "A dual-mode NAND flash memory: 1-Gb multilevel and high-performance 512-Mb single-level modes," *Solid-State Circuits, IEEE Journal of*, **36**(11), pp. 1700–1706.

[77] Qureshi, M. K., M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis (2010) "Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, ACM, New York, NY, USA, pp. 153–162.
URL http://doi.acm.org/10.1145/1815961.1815981

[78] Lee, S., K. Ha, K. Zhang, J. Kim, and J. Kim (2009) "FlexFS: A Flexible Flash File System for MLC NAND Flash Memory," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, USENIX Association, Berkeley, CA, USA, pp. 9–9.
URL http://dl.acm.org/citation.cfm?id=1855807.1855816

[79] Grupp, L. M., J. D. Davis, and S. Swanson (2013) "The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, USENIX Association, Berkeley, CA, USA, pp. 79–90.
URL http://dl.acm.org/citation.cfm?id=2535461.2535472

[80] Jimenez, X., D. Novo, and P. Ienne (2013) "Phoenix: Reviving MLC blocks as SLC to extend NAND flash devices lifetime," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 226–229.

[81] Azevedo, R., J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin (2013) "Zombie Memory: Extending Memory Lifetime by Reviving Dead Blocks," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, New York, NY, USA, pp. 452–463.
URL http://doi.acm.org/10.1145/2485922.2485961

[82] Dong, X. and Y. Xie (2011) "AdaMS: Adaptive MLC/SLC Phase-change Memory Design for File Storage," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, IEEE Press, Piscataway, NJ, USA, pp. 31–36.
URL http://dl.acm.org/citation.cfm?id=1950815.1950821

Vita

Ellis H. Wilson III

Education

The Pennsylvania State University, University Park, PA, US. Ph.D., Computer Science and Engineering, August 2014.
La Salle University, Philadelphia, PA, US. B.S., Computer Science, May 2009.

Honors and Awards

- Supercomputing: Best Paper Finalist & Best Student Paper Finalist, 2013
- Supercomputing: 2nd Place, ACM Student Research Competition, 2012
- The Pennsylvania State University: College of Engineering Fellowship, 2009
- NCAA Division I Cross Country: Atlantic 10 All-Conference, 2007
- NCAA Division I Cross Country: Atlantic 10 Academic All-Conference, 2007
- Boy Scouts: Eagle Scout, 2005

Publications

- **Ellis H. Wilson III**, Mahmut T. Kandemir, Garth Gibson. "Will They Blend?: Exploring Big Data Computation atop Traditional HPC NAS Storage." *International Conference on Distributed Computing Systems*, 2014.
- **Ellis H. Wilson III**, Myoungsoo Jung, Mahmut T. Kandemir. "ZombieNAND: Resurrecting Dead NAND Flash for Improved SSD Longevity." *International Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems*, 2014.
- Myoungsoo Jung, Wonil Choi, Shuwen Gao, **Ellis H. Wilson III**, David Donofrio, John Shalf, Mahmut Taylan Kandemir. "NANDFlashSim: High-Fidelity, Micro-Architecture-Aware NAND Flash Memory Simulation." *ACM Transactions on Storage.* In Submission.
- Myoungsoo Jung, **Ellis H. Wilson III**, Wonil Choi, John Shalf, Hasan Metin Aktulga, Chao Yang, Erik Saule, Umit V. Catalyurek, Mahmut Kandemir. "Exploring the Future of Out-Of-Core Computing with Compute-Local Non-Volatile Memory." *Supercomputing*, 2013.
- **Ellis H. Wilson III**. "Performing Cloud Computation on a Parallel File System." *ACM Student Research Competition at Supercomputing*, 2012.
- Myoungsoo Jung, **Ellis H. Wilson III**, Mahmut Kandemir. "Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks." *International Symposium on Computer Architecture*, 2012.
- Myoungsoo Jung, **Ellis H. Wilson III**, David Donofrio, John Shalf, Mahmut Kandemir. "NANDFlashSim: Intrinsic Latency Variation Aware NAND Flash Memory System Modeling and Simulation at Microarchitecture Level." *IEEE Conference on Massive Data Storage*, 2012.