Hardware-Driven Evolution in Storage Software

by

Zev Weiss

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: June 8, 2018

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Michael M. Swift, Professor, Computer Sciences

Karthikeyan Sankaralingam, Professor, Computer Sciences

Johannes Wallmann, Associate Professor, Mead Witter School of Music

To my parents,
for their endless support,
and my cousin Charlie,
one of the kindest people I've ever known.

Acknowledgments

I have taken what might be politely called a "scenic route" of sorts through grad school. While Ph.D. students more focused on a rapid graduation turnaround time might find this regrettable, I am glad to have done so, in part because it has afforded me the opportunities to meet and work with so many excellent people along the way.

I owe debts of gratitude to a large cast of characters:

To my advisors, Andrea and Remzi Arpaci-Dusseau. It is one of the most common pieces of wisdom imparted on incoming grad students that one's relationship with one's advisor (or advisors) is perhaps the single most important factor in whether these years of your life will be pleasant or unpleasant, and I feel exceptionally fortunate to have ended up

with the advisors that I've had. I have always been granted plenty of independence, but also given the guidance and suggestions to point me in a useful direction when I've gotten stuck. Andrea's thorough, thoughtful feedback on paper drafts (and of course this very document) has been invaluable. Her CS402 program was a rewarding, enjoyable experience (even though I unfortunately missed the semesters when she was actually around running it herself), and provides a wonderful service to Madison schools and their students. I am glad Remzi happened to notice my shenanigans in 537 projects (and deem them acceptable, even when they were horrifying), and in spite of them still provide me the opportunity to teach the same course some years later. I will miss our weekly meetings, especially the ones that veered off course.

To Ed Almasy and Rachael Bower, fearless co-directors of the Internet Scout Research Group, who welcomed me into their wonderful work environment, where I happily remained through nearly my entire career as a grad student.

To Rustam Lalkaka, officemate and co-sysadmin at Scout, who perhaps unintentionally ended up responsible for a large part of how the rest of my time here at UW-Madison has unfolded by suggesting that I take CS537 from Remzi. ("Who?" I said. "He's cool, you won't regret it", he replied, prophetically.)

To Corey Halpin, with whom I have shared countless lengthy and enjoyable conversations on many matters, but most often a shared (excellent!) taste in software.

To Johannes Wallmann, for guiding me through my music minor, running an excellent jazz program at the UW School of Music, and bravely serving on a CS dissertation defense committee!

To Karu Sankaralingam and Mike Swift, for the interesting courses they've taught (and taught well), and for agreeing to listen to me defend this dissertation.

To Tyler Harter – the only student coauthor I've worked with in my time here, but as excellent a coauthor as one could hope for, with a remarkable knack for presenting complex topics in clear, comprehensible ways. To Jun He, who has endured me as an officemate for longer than anyone else, providing numerous interesting discussions along the way. And to all the other members of the Arpaci-Dusseau group I've worked with and learned so much from over the last seven years: Ram Alagappan, Leo Arulraj, Vijay Chidambaram, Thahn Do, Aishwarya Ganesan, Joo Yung Hwang, Sudarsun Kannan, Samer al-Kiswany, Jing Liu, Lanyue Lu, Yuvraj Patel, Thanu Pillai, Kan Wu, Suli Yang, Yiying Zhang, Yupu Zhang, and Dennis Zhou.

To the friends I've made in the department here: Ben

Bramble, Mark Coatsworth, Adam Everspaugh, Thomas Griebel, Rob Jellinek, Kevin Kowalski, Ben Miller, Evan Radkoff, Will Seale, Brent Stephens, Venkatanathan Varadarajan, Ara Vartanian. The diverse discussions, project collaborations, beers on the terrace, and other adventures many and varied have been a pleasure.

To the people I worked with at Fusion-io: Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, and the members of the Clones team. I thoroughly enjoyed my time there, and am grateful for the honor of being granted, via custom-emblazoned sweatshirt, the status of "intern emeritus" when I left to return to Madison.

To the many people of SimpleMachines, where there's never a dull moment, and where almost everything is good.

To Angela Thorpe for being so helpful with administrative questions and deftly coordinating the CS graduate program – especially critical for those among us who perhaps fall slightly toward the less-organized end of the spectrum.

To Eric Siereveld for skillfully directing the UW Latin Jazz Ensemble the two years I was fortunate enough to play in it, and Josh Agterberg, Andrew Baldwin, Rachel Heuer, and Will Porter for providing a wonderful ensemble to perform with for my minor recital.

To Michaela Vatcheva, for so many interesting times and

conversations, and for getting me to (after sufficient prodding) finally join the sailing club – my only regret is that I took so long to heed this advice.

To Amy De Simone for befriending me when I had just moved to a new and unfamiliar city, and bringing me along on walks with her dog.

To Becky, for her patience and encouragement, especially in these last few weeks.

And to my sister, Samara, and parents, Alan and Cheryl, for their love and support.

Contents

List of Figures

Abstract

As technology improves, changes in hardware drive corresponding adaptations in software. This thesis examines the hardware-driven evolution of both applications and system software as they relate to the matter of data storage in modern computing systems.

The move from single-processor systems to ones with numerous CPU cores executing in parallel has motivated applications to make increasing use of multithreading. The resulting nondeterminism introduces new difficulties to the common technique of evaluating storage system performance by replaying traces of application execution. We present the ROOT technique and implementation of it, ARTC, to address this challenge and provide a trace replay system for

multithreaded applications that is both reliable and accurate in its performance projections.

Storage hardware has also undergone major changes in recent years, with traditional hard-disk drives increasingly displaced by flash-based SSDs, and even more recently emerging nonvolatile memory technologies. This shift drives the need for new software to manage these new devices and provide useful storage features and functionality, such as file cloning and deduplication, in a manner well-suited to the characteristics of the new hardware. Here we present ANViL, a storage virtualization system that provides these features in a novel way developed expressly for flash storage.

The dramatic difference in the performance characteristics of emerging storage technologies relative to the much slower mechanical devices they are replacing, however, also shines a new and unflattering light on the performance of storage software. Much of this software dates from the era of the hard-disk drive, when CPU cycles were often considered essentially "free" in comparison to the long latencies of disk operations. This performance imbalance made it easy to do relatively expensive things in software, safe in the knowledge that their performance cost would be hidden by the much slower storage devices they managed. However, as the performance gap between CPUs and storage hardware narrows, the

CPU execution performance of software in the storage stack becomes increasingly critical. For this problem we present DenseFS, a prototype filesystem with the explicit aim of minimizing its use of CPU cache resources in an effort to not only run efficiently itself, but also to reduce its impact on application performance.

These pieces exemplify how software evolution in response to changes in hardware occurs, but also how it differs as the hardware in question becomes increasingly well-established. Multicore CPUs have been commodity items for over a decade and are now nearly unavoidably ubiquitous; we examine a delayed, second-order effect of this change on a specialized area of storage software, as its more immediate effects have been studied since it was a younger technology. Flash has been widespread for some time, but is not yet so deeply ingrained in the hardware landscape; here we examine one part of the ecosystem of storage software that is still in the process of adapting to suit the new hardware. Finally, NVM technologies are just beginning to arrive; the major, first-order questions it raises, such as what an NVM-oriented filesystem might look like, are thus still being addressed. Taken together then, these three components illustrate different stages in the chronology of how software's hardware-driven evolution has occurred, and how we expect it is likely to continue.

1 Introduction

As computing hardware evolves over time, its interfaces typically maintain backward compatibility so as not to disrupt the operation of existing software. In order to fully exploit the potential of improved hardware, however, software in both applications and operating systems must also adapt. This thesis explores such hardware-driven software evolution in the specific context of storage systems.

1.1 Trace Replay in the Multicore Era

As clock frequencies and serial CPU performance have gradually plateaued, performance improvements in recent generations of CPUs have come largely from increasing parallelism

in the form of ever-growing numbers of CPU cores [13]. Off-the-shelf, consumer-grade desktops and laptops in 2018 are typically equipped with between two and eight CPU cores; servers often offer dozens. In order to make use of this added processing power, application code for tasks that would previously have been implemented with fewer threads (or perhaps only one) has evolved to employ increasing numbers of threads [106].

The adoption of multithreading in applications has introduced significant new questions and problems, including the difficulty of writing programs that avoid race conditions and deadlocks [32, 43, 124, 133], how to debug multithreaded programs effectively [18, 85, 103], and how to avoid performance bottlenecks that limit scalability [19, 24]. Specific incarnations of these problems, particularly regarding issues of performance and scalability, have also arisen in the area of storage systems, and have been the subject of prior research [20, 100, 109, 160].

Here we examine a second-order effect of multithreaded applications on storage systems, specifically in their interactions with trace replay, a popular and useful technique commonly used in evaluating the performance of storage software and hardware. The nondeterministic behavior of multithreaded applications poses a problem for trace replay: the behavior of the application (particularly in the ordering of its filesystem

operations between different threads) can be highly dependent on the particular performance characteristics of the system on which it runs. This dependency means that a simple trace replay that does not exhibit nondeterminism similar to that of the original application itself can result in behavior that diverges substantially from that of the actual application running on the same system.

We explore this problem in detail and present a novel technique, Resource-Oriented Ordering for Trace replay (ROOT), to address it by safely preserving the nondeterminism of multithreaded applications during replay. The ROOT approach uses automated analysis of the events recorded in a trace to examine the set of resources accessed by each event and then construct a graph of inter-event dependencies. Since we apply this technique to Unix system calls (and particularly filesystem operations), we focus on filesystem-related resources, such as files and path names. The resulting dependency graph can then be used during replay to allow it to perform actions in different orders than the one recorded in the trace, preserving the multithreading nondeterminism of the original program, while still maintaining the semantics of the original ordering in the trace.

We present an implementation of ROOT called ARTC (an Approximate-Replay Trace Compiler) that performs nondeter-

ministic replay of multithreaded Unix system call traces across a variety of Unix-like operating systems. We evaluate ARTC in two key areas, its semantic correctness and its performance accuracy, and compare it to three simpler approaches to the problem, one which reorders more freely than ARTC and two that conservatively disallow any reordering.

To evaluate semantic correctness, we use ARTC to replay a suite of complex traces of modern multithreaded desktop applications [56] and measure its error rate (the number of deviations from the operation results recorded in the original trace). We find that ARTC achieves a nearly identical degree of semantic correctness to the order-preserving replay modes, while the less constrained mode often fails catastrophically.

We then continue our evaluation using a series of micro- and macro-benchmarks to measure ARTC's performance accuracy – specifically, how closely it matches the performance of the original program when system parameters are changed. To illustrate feedback effects between systems and applications (where the performance characteristics of the system affects the behavior of applications running on it), we run a set of microbenchmarks, each aimed at a specific parameter. In each case, ARTC responds appropriately as the parameter is adjusted, accurately tracking the performance of the original program re-executed with the same adjustment, deviating by

at most 5%. In contrast, the other replay strategies we evaluate frequently yield wildly inaccurate performance estimates, often erring by 15-50%, and in some cases far more. We then extend our performance accuracy evaluation with two LevelDB workloads. With seven system configurations, we evaluate the full cross-product with one configuration as the trace source and another as the replay target, comparing performance of the trace replay against the original program on the target system. Here we again find that ARTC performs much more accurately than other replay methods, achieving a median timing discrepancy of 7.6%, with the median inaccuracy of next closest method being 19.1%.

By embracing the nondeterminism of multithreaded applications, we have demonstrated a trace replay methodology that improves on the state of the art. Our replay uses careful analysis to permit safely-constrained reordering, allowing it to achieve superior replication of the behavior and performance of real applications. ROOT and ARTC thus help trace replay for storage systems to adapt to the requirements of the multicore era.

1.2 Advanced Virtualization for Flash Storage

For many years hard-disk drives (HDDs) were the dominant technology in storage hardware [8]. During this period, a great deal of software in the storage stack was developed around the particular characteristics of hard disks, such as filesystems and databases with layouts optimized for the specifics of disk geometries [92, 96, 98] and I/O scheduling algorithms designed to maximize the throughput of read/write heads seeking across spinning platters [63, 69, 127, 149].

In the last decade, however, hard disks are being broadly supplanted by solid-state storage devices (SSDs) with fundamentally different characteristics [50, 83, 102]. NAND flash, to take one common example, cannot be directly overwritten; it requires that a large contiguous block of data be erased before any data within it can be rewritten, and each such program/erase cycle puts physical wear on the storage chip itself, shortening its remaining useful life [102]. To compensate for added complexities such as these, however, solid-state storage offers far lower access latencies than HDD storage.

The advent of a new storage technology that both introduces a significant leap in performance and changes the nature of what constitutes a desirable I/O pattern presents

a problem for storage software. As long as storage interfaces are maintained in a backwards-compatible manner, existing software components continue to function, and do achieve performance gains, but as they are designed primarily around the parameters of an entirely different family of hardware devices they are unlikely to be optimally suited for the newer hardware technology they are suddenly paired with. This mismatch drives a need for redesigned software that takes into account the characteristics of the new hardware to better utilize its performance potential and avoid premature device wear-out [23, 71, 79, 81].

With ANViL we present such a redesigned component of the storage stack. ANViL is a block-level storage virtualization system designed for modern high-performance flash hardware. It builds upon its basic underlying flash-friendly structure to provide a feature set that extends the conventional block-I/O interface with a small set of new operations that offer a great deal of added power while remaining simple and easy to integrate into existing systems.

ANViL is based on a log-structured, redirect-on-write design; the added complexity and expense of direct over-writes in flash storage make this an advantageous strategy for flash-oriented storage systems [79, 155, 159]. It extends its address-translation layer, however, to allow a many-to-one

address map and allows filesystems and applications running above it to directly manipulate ranges of this map with clone, move, and delete operations that augment the conventional read/write block storage interface. We details these range operations and the implementation of ANViL, going into particular depth regarding its garbage collector (GC).

The ANViL GC faces a more challenging problem than the GCs of most log-structured storage systems due to the many-to-one nature of the address map in combination with the scale and performance levels at which ANViL is target to operate. To address this challenge we have designed and implemented a specialized GC comprised of two components, the *scanner* and the *cleaner*, counterparts to the mark and sweep phases often used in tracing GCs in programming-language implementations [147], the unlike a mark-and-sweep collector, our scanner and cleaner operate concurrently and continuously instead of in serialized phases. The GC incorporates a variety of optimizations and specialized implementation techniques, including multithreading with dynamic work partitioning, pipelining, and a hook mechanism analogous to the write barriers used in programming-language GCs. Our evaluation demonstrates that the GC can keep up with the demands placed on it by heavy foreground write traffic, and exhibits near-linear performance scaling as data quantities

and the population of the address map increase.

After detailing ANViL's implementation, we then demonstrate the utility of the extension to the standard block interface provided by its range operations. We show how the clone operation can be used to, with only a few hundred lines of code, implement support for low-cost file snapshots and deduplication in ext4 – a relatively conventional update-in-place filesystem not designed with such features in mind. We also demonstrate how the same operation can be easily used to provide volume snapshots in the style of LVM [58], but with a far smaller penalty on the volume's post-snapshot I/O performance. As a final case study, we show how ANViL's range move operation can be used to implement a powerful transactional-commit mechanism that eliminates the usual cost of writing data twice, and can be easily incorporated into existing transactional systems such as journaling filesystems; we demonstrate by integrating it into ext4's jbd2 journaling layer. In addition to improving performance and reducing wear on flash cells by avoiding the transactional double-write penalty, utilizing this mechanism can even simplify the surrounding filesystem code by eliminating the need for complex recovery procedures.

We have designed ANViL's structure and I/O patterns to mesh well with the fundamental characteristics of NAND

flash devices. The address-remapping layer that is central to its flash-friendly design can be exposed via a set of small extensions to the block-I/O interface to enable the storage virtualization layer to provide a new dimension of functionality to applications and filesystems. ANViL thus provides an example of how storage virtualization can be updated to be better matched to the solid-state storage hardware that is now widespread.

1.3 Cache-Compact Filesystems for NVM

While the relatively high-performance flash SSDs targeted by ANViL have taken over a large (and still-growing) fraction of the storage-hardware market, it appears that the landscape of storage hardware may be about to undergo another major shift with the arrival of nonvolatile memory (NVM) in the form of technologies such as phase-change memory and memristors [45, 54, 130]. These devices offer the persistence of hard disks or SSDs, but provide a memory-like interface (operating via simple load and store CPU instructions as opposed to the block I/O interface used by most existing persistent storage devices) and access latencies closer to those of DRAM than of existing persistent storage devices. While actual NVM

hardware has only recently become available and hence has not thus far seen widespread adoption, the questions of its integration into storage systems has nevertheless been the subject of research in the last few years.

The drastically reduced access latencies of NVM invert the performance assumptions of existing storage software to an even greater degree than did the arrival of SSDs. Current filesystem designs targeted at the relatively high performance of flash in comparison to magnetic disks may still impose excessive software overhead, rendering the system incapable of utilizing the full performance of its storage hardware. Research efforts to address this problem have considered major restructuring of storage software, such as moving filesystems into user-level code, or even into storage devices themselves [27, 74, 111, 142, 144].

Another aspect of NVM hardware to which existing software is not well-matched stems from its byte-addressable, memory-style interface. Hard disks and SSDs provide an interface by which software can perform read and write operations in relatively large, fixed-size units (typically 512 or 4096 bytes), and guarantee the atomicity of individual block writes in the event of a power loss. NVM, in contrast, is accessed by software in the same manner it addresses regu-

lar (volatile) memory: byte by byte.[1] This interface change presents an incompatibility with the consistency mechanisms in current storage stacks, which are often reliant on the larger atomic-write capability provided by block-oriented I/O devices [97, 113, 137]. A variety of methods to solve this problem for filesystems have been the subject of research in recent years [39, 150, 152], as have techniques to address analogous problems that arise in application code [21, 35, 139].

We examine a specific aspect of filesystems relating to the high performance of NVM hardware. As latencies decrease, storage-intensive applications that had previously been bottlenecked by the relatively slow performance of storage hardware may instead find CPU time an increasingly limiting factor on their performance. This inversion makes both the raw CPU execution performance of storage software and its effects on the performance of application code suddenly much more critical.

With this in mind, we first analyze the behavior of current filesystems with regard to one of the most critical hardware resources for execution performance: the CPU cache. We perform a detailed study of the cache footprints and access patterns of five existing Linux filesystems (btrfs, ext4, f2fs, xfs, and tmpfs). Using instruction-level dynamic execution tracing

[1] Though the NVM hardware itself, sitting outside the CPU's cache hierarchy, will see cache-line-granularity accesses.

of the end-to-end kernel code paths of various filesystem operations, we gather data for both instruction and data memory accesses and construct detailed visualizations of the results, finding that most filesystem operations have data cache footprints that displace most of the first-level cache state in current CPUs, and even larger instruction cache footprints that often exceed the size of the L1 instruction cache by 50% or more. Further, a large fraction of these cache footprints, especially in the instruction cache, see little to no reuse of the accessed cache lines, indicating inefficient use of the cache hardware (which optimizes for access patterns that exhibit a greater degree of temporal locality).

In order to gain a better high-level understanding of the sources of the code footprints of these operations, we condense the detailed source-level stack traces collected at each instruction in our traces into coarse-grained stack traces that indicate the provenance of each instruction in terms of major categories of code (such as memory allocation, the page cache, or journaling) instead of by individual lines of source code. In examining the resulting data, we see that major, "unavoidable" components (those common to all filesystems), such as the VFS layer and page cache, are responsible for a substantial fraction of overall code footprint across all filesystems.

With this knowledge, we then proceed to implement and

evaluate a filesystem design, DenseFS, that makes optimization of its cache usage its primary goal. DenseFS aims to not only achieve high performance execution of its own filesystem operations, but also to improve the execution performance of application code by reducing the cache pollution incurred by its operations. To avoid some of the major sources of increased code size, we begin by implementing DenseFS outside of the usual framework of kernel components in which filesystems typically operate (the VFS, page cache, etc.), instead introducing a set of DenseFS-specific system calls. We describe a variety of techniques we have employed to further compact its code and data structures byte by byte and cache line by cache line. The result is a highly compact filesystem; the total code footprints of its operations in most cases occupy less space than the code footprints of the VFS code alone for the same operations in the existing filesystems we evaluate.

We then evaluate the performance of DenseFS using a microbenchmark and a recursive `grep`, each using DenseFS's mirror set of system calls instead of the standard filesystem operations such as `open`, `read`, and `stat`. Our microbenchmark enables precise measurements of the impact of filesystem operations on the CPU performance of user code with varying code and data working-set sizes. We use it to measure the performance cost incurred by incorporating filesystem

operations into otherwise system-call-free code, and find that the performance impact of DenseFS is in almost all cases much lower than any other filesystem, often incurring only a 10-20% loss where other filesystems cause degradations of 50-150%. Our experiments with `grep` show large reductions in kernel execution times as well as improvements of 13-18% in user-mode CPU performance.

This version of DenseFS, however, suffers from a severe practical drawback in requiring applications to use a special set of dedicated DenseFS system calls. To address this shortcoming, we then implement DenseFS2, which is slightly more integrated into the rest of the kernel – just enough to be accessed via the existing standard file-access system calls, but still avoiding the bulk of the VFS layer by quickly detecting DenseFS2 operations and shunting control to it early in the call relevant code paths. This arrangement makes evaluating performance with other applications much easier, as they no longer require any special treatment to access their data via DenseFS2. We are then able to perform further performance evaluation with an unmodified SQLite benchmark program. On this workload DenseFS2 reduces overall execution time by 20-80% across all configurations we evaluate, and increases user-mode CPU performance (IPC) by 9-82%.

While it is currently a prototype filesystem with an array

of practical difficulties, DenseFS clearly demonstrates the importance of filesystem cache behavior to overall performance. With appropriate refinement, we hope that some of its ideas might one day help filesystems for low-latency NVM devices better exploit the performance potential of their hardware resources.

1.4 Overview

Here we provide a broad summary of the contents of the following chapters so as to provide an overview of the research presented in the remainder of this dissertation.

In Chapter 2 we present our work in the area of multi-threaded trace replay with ROOT and ARTC. We explain the principles of the ROOT analysis and replay methodology and present the implementation of our ROOT-based trace replay tool, ARTC, and the results of our evaluation of ARTC's correctness and performance. My own contributions to this work are the constrained nondeterministic replay approach formulated as ROOT, the development of the ARTC replay system, and, in collaboration with Tyler Harter, the evaluation of ARTC.

In Chapter 3 we present ANViL, our flash-oriented storage virtualization system. We detail ANViL's design and organi-

zation, with particular attention to the challenging problem of its garbage collection. We demonstrate, with a series of case studies, how its range operations can be easily integrated into existing software to provide a variety of useful features, and evaluate the scalability of its garbage collector. I contributed the development of the ANViL garbage collector, assistance with the design and implementation of other parts of ANViL (developed by the Advanced Development Group at Fusion-io and later SanDisk), and the evaluation of the system presented herein.

In Chapter 4 we present our cache-optimized, NVM-targeted filesystem, DenseFS. We begin with our trace-based analysis of existing filesystems; we then present the design and implementation of DenseFS, as well as a second-generation version that addresses the primary shortcoming of the first by integrating into existing system calls. We evaluate these implementations with an assortment of targeted microbenchmarks and application programs. The design, implementation, and evaluation of DenseFS presented here are my own individual work.

Finally, Chapter 5 concludes with a summary of the contributions of this dissertation and some discussion of possible future research that could extend the work presented here.

2

Accurate Trace Replay for Multithreaded Applications

Trace replay is an important tool in the systems researcher's toolbox. In instrumenting a running system to collect a detailed record of its actions (a trace) and then later synthesizing the execution of those same recorded actions (replaying the trace), we have a useful technique by which we can reproduce a system's behavior in a controlled environment for purposes such as performance analysis, optimization, and debugging. Much existing work in the area of trace replay, however, operates under the assumption that the most faithful reproduction of the behavior of the original system is one that replays recorded actions in the trace exactly as they appear therein. While this assumption is intuitively reasonable and

may be entirely correct in simple cases, it begins to break down when applied to more complex systems, particularly those that exhibit nondeterministic behavior.

With the rise of ubiquitous multicore CPUs, software has begun to increasingly incorporate multiple threads in order to take advantage of the available hardware. Multithreading is, however, one of the best-known sources of nondeterminism in computing systems, which complicates the problem of trace replay. Tracing two runs of the same multithreaded program on the same system is highly likely to produce two different-looking traces, as the actions performed by multiple concurrently-executing threads are interleaved in different orders. Traces taken from the same program on distinct systems (for example, ones with different performance characteristics) will likely diverge even more. Replaying such a trace strictly as it was recorded, then, will not accurately reproduce the naturally nondeterministic behavior of the original program.

In this chapter we devise a new trace replay methodology, ROOT, to address this problem, and present an implementation of this methodology called ARTC. We describe the details of how ROOT allows some reordering of actions recorded in a trace, but constrains this reordering via resource-based analysis so as not to violate the semantics of the trace. We then evaluate ARTC in comparison to some less sophisticated

methods of trace replay and find that it achieves a high degree of semantic correctness, while also providing a much better reproduction of the original program's performance characteristics than do the simpler replay methods.

2.1 Introduction

Quantitatively evaluating storage is a key part of developing new systems, exploring research ideas, and making informed purchasing decisions. Because running actual applications on a variety of storage stacks can be a painful process, it is common to collect statistics or traces on a single system in order to understand an application [16, 42, 64, 82, 107, 120, 126, 146, 153].

Trace replay is a useful technique for evaluating the performance of different systems [9, 72, 76, 84, 95, 105, 135]. Here we focus particularly on the use of trace replay for performance *prediction*. A trace of a running application may be collected on one system (the *source*) and replayed on another (the *target*) in order to predict how the original application would perform on the target system. Trace replay can be a valuable tool in evaluating potential system changes such as upgrading hardware, switching to a different filesystem, or simply adjusting a configuration parameter.

There are a variety of points within the storage stack at which traces can be collected and replayed. These are typically at well-defined interfaces between different software or hardware components. For example, in a distributed storage system, one might trace requests as they arrive at a server via the network [41], and replay the resulting trace by generating network packets encoding the recorded requests and sending them to the server from a synthetic client application. Tracing and replay of program behavior can be performed at the system call interface – the boundary layer between applications and the operating system [99]. Another major boundary layer in the storage stack, the block interface at which filesystems issue raw I/O requests to their underlying storage devices, can also be used as point of introspection for tracing [135]; replay in this case is typically performed by a specialized application accessing a block device directly instead of via a filesystem.

Different interfaces in the storage stack have advantages and disadvantages as potential points for trace collection and replay. As a general rule, tracing at a given point allows replay of those traces to be used in evaluating changes in components that sit "downstream" of that point in the stack. To illustrate in the context of the examples above, a trace of network requests made to a server in a distributed system

could be used to evaluate any software or hardware component within the server, because all of those components may play a role in servicing the requests recorded in the trace. In the same system, a trace of the system calls made by the process on the server receiving and servicing those requests could be used in evaluating changes to the local filesystem or storage hardware on the server, but not, for example, a change to the server process itself that allows it to service some requests from an in-memory cache without accessing the local filesystem, because the system call trace has already captured (and thus frozen) that aspect of the server process's behavior. Continuing downward, a block-level trace additionally captures the behavior of the filesystem and page cache, and would thus not be useful in measuring the performance of a different filesystem or the effects of dedicating more RAM for use in the page cache, but could still be used to evaluate changes in the block layers of the storage system, such as the relative performance of different RAID array configurations.

Performing tracing and replay at higher points in the storage stack is thus appealing in broadening the scope of the underlying components that such traces can be used to evaluate. Higher-level traces are not without their downsides, however, as moving the traced interface higher in the stack also constrains the applicability of the traces. A network-request-

level trace of a distributed system would only be applicable for replay in the context of a distributed storage system (and in practice, given the specificity of network protocols, likely only that specific system). Similarly, a system call trace from a Unix-like operating system would be of little use in attempting to replicate the behavior of the application by replaying it on Windows (or would at least require substantial amount of additional development effort for the replay to "translate" the operations in the trace to match the semantics of a different system call API). In this regard, the relatively low-level block layer trace is highly general, as the same simple, standard block storage interface is ubiquitous across the overwhelming majority of widely-deployed operating and storage systems.

We find that replaying traces at the system-call level provides the best balance of the breadth of contexts in which it is applicable and the scope of the system components it can be used to evaluate. While the Unix system call interface does not encompass all the world's operating systems (and there exist subtle variations even among nominally Unix-like systems, as detailed in Section 2.4.3), it is a reasonably standardized, consistent API in wide use across a variety of segments of the computing industry, being popular on server, desktop, and mobile platforms. It is thus applicable to a multitude of applications, while also sitting atop a rich system

of underlying components that it can be used to measure.

At first it might seem that trace replay would offer easy insight into an application's performance on an alternate storage stack, since the actions replayed are precisely the actions the real application performed. However, this glosses over the nondeterminism of multithreaded applications, which have become increasingly prevalent with the advent of multicore CPU hardware. In such applications, while the ordering of operations within each individual thread may be fixed, there is no single fixed global order of operations across all threads. Even on the same system, two executions of the same program may produce slightly differing orderings of events; when comparing across different systems the likelihood of two runs of a multithreaded program issuing system calls in the exact same global order becomes vanishingly small [47, 99].

Furthermore, a complex feedback relationship exists between applications and the systems on which they run: the behavior of each both affects and depends on the behavior of the other. The ordering and timing of the requests issued to the system by the application affects the manner in which the system performs them, but the ordering and timing of system's completion of those requests then also affects the manner in which the application issues subsequent requests.

For a simple example of the effects of feedback between

systems and applications, consider an application with two threads, each of which independently performs two consecutive synchronous reads. If Thread 1 and Thread 2 issue their first reads concurrently, one system might complete Thread 1's read first, allowing it to issue its second read well before Thread 2 can request its own second read; another system running the same program may instead complete Thread 2's first read before Thread 1's, resulting in the opposite ordering of the second read from each thread. Extrapolating effects of this nature throughout the entire execution of large programs with many threads, it is clear that realistic replay of multithreaded traces is complex, and simplistic approaches that adhere too closely to the exact behavior recorded in a given trace will not accurately reflect the actual behavior of real programs across different systems.

In evaluating the quality of different approaches to trace replay for performance prediction, we use two main criteria: *semantic correctness* and *performance accuracy*. The former measures how well the semantics of the operations recorded in the trace are reproduced by the replay; the latter measures how close the replay's performance on the target system predicts that of the original program.

In some trace replay scenarios, semantic correctness is nearly trivial; for example, there is little difficulty in replicat-

ing the semantics of a single sequential stream of block-I/O requests. With system-call replay, however, semantic correctness is less simple: files of appropriate sizes must exist at appropriate locations, possibly with extended attributes and other metadata correctly initialized. Considering multithreaded traces with the possibility of system calls being reordered between threads introduces further complexity: if an `open` and a `read` in two different threads are reordered with respect to each other, leading the `read` to attempt to access data from a file that is not yet open, the `read` may fail with `EBADF`, deviating from the semantics of what occurred in the original application.

Trace-replay tools should reflect the characteristics of applications, including the ordering dependencies of their execution. Two types of artifacts can provide information about the dependencies of an application: the original program itself, and traces of its execution. Unfortunately, application source code is often unavailable, and deriving full, application-level semantic dependencies from a single trace collected on one system is generally not possible. However, the ways in which programs manage storage resources, as recorded in a trace, can provide hints about a program's dependencies. We propose a new technique for extracting these hints from a trace and utilizing them for replay: Resource-Oriented Ordering for

Trace replay (ROOT). The ROOT approach is to observe the ordering of the actions that involve each individual resource in a trace and apply a similar ordering to the corresponding actions during replay.

We have built a new tool, ARTC (an "Approximate-Replay Trace Compiler"), that implements the ROOT approach to replaying system-call traces of multithreaded applications. ARTC constrains replay based on resource-management hints extracted from a trace. In order to extract meaningful hints, ARTC uses a detailed Unix filesystem model and knowledge of over 80 system calls to infer the complex relationships between actions and resources. For example, awareness of symbolic links allows ARTC to track all of the pathnames that refer to a given file resource; similarly, a directory-tree model allows ARTC to determine the entire set of resources that are affected by directory rename operations.

We use ARTC to automatically generate a new cross-platform benchmark suite, Magritte, from 34 traces of Apple desktop applications [56]. Because many of these traces contain OS X-specific system calls, we employ novel emulation techniques for 19 different calls, allowing replay of the traces on other systems.

We compare ARTC against three simpler replay strategies: a single-threaded approach, a multithreaded replay that

disallows reordering, and an unconstrained multithreaded replay with no synchronization between threads. We use the complex Magritte workloads to evaluate semantic correctness, finding that ARTC achieves error rates nearly identical to those of the more heavily constrained replays. For timing accuracy, we demonstrate the weaknesses of the simple replay methods with microbenchmarks designed to illustrate behavioral feedback effects in the storage stack involving workload parallelism, disk parallelism, cache size, and I/O scheduling. In these experiments we show that simple replay methods can produce highly inaccurate performance predictions, in some cases estimating execution times as low as 19% and as high as 705% of those of the original program on the same system. We also replay traces of an embedded database, and find that ARTC reduces average timing error from 21.3% (for the most accurate alternative) to 10.6%.

2.2 Trace Mining

We now consider what types of information can be extracted from traces for the purpose of replay. A single trace provides a sequence of actions in a certain order that the program may generate when run on a specific system with a given set of inputs. Ideally, however, we would like to infer the entire

space of action orderings that the program *could* produce when run with those same inputs. We refer to this as the *I/O space* of a particular combination of a program and its input. For example, a trace of simple program might provide the following ordering of actions:

```
create directory "/a"
open file "/b" as file descriptor 3
read 512 bytes from file descriptor 3
close file descriptor 3
```

This particular trace represents a single known-valid point in the I/O space of that program with that input. It is however possible that the same program run with the same input on a different system (or even simply in another execution on the same system) might, for example, open `"/b"` *before* creating the directory `"/a"` instead of after doing so; this would be a manifestation of another point in its I/O space. To achieve accurate replay we aim to infer points in the I/O space beyond those presented in the available trace data.

Depending on the type and quantity of the available traces, different techniques may be used to infer the I/O space, and different degrees of accuracy will be achievable. We now define various types of trace data that may be available (Section 2.2.1)

and describe three inference techniques, including our new technique, ROOT (Section 2.2.2).

2.2.1 Trace Inputs

There are three key attributes of parallel trace data: the number of traces, whether the collection of the traces was active or passive, and whether or not the traces include synchronization information.

First, some inference techniques require many traces. Each trace represents one point in the I/O space of the application; observing many points makes it easier to guess the shape of the whole space. Unfortunately, collecting many traces on the same system will tend to explore only certain areas of the whole I/O space.

Second, traces may be collected either passively or actively. Passive tracing simply records an application's I/O actions, doing nothing to interfere. In contrast, active tracing may perturb I/O; certain operations may be artificially slowed so as to observe the resulting effects on the timing of subsequent I/O actions. The active method thus allows direct deduction of dependencies and methodical exploration of the I/O space.

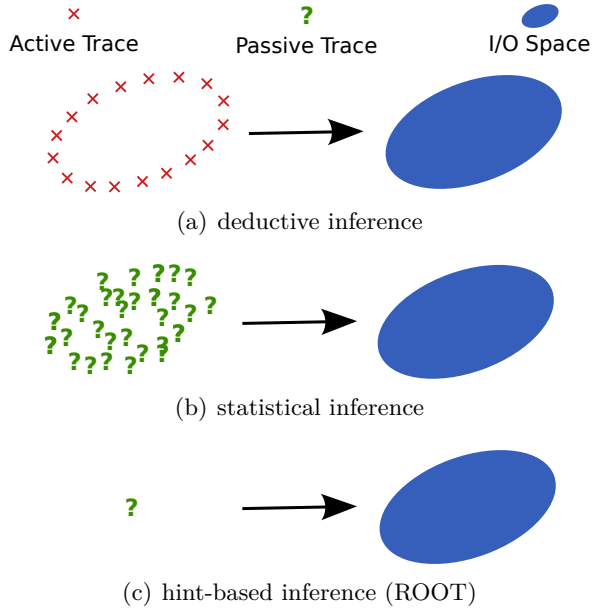
Third, traces may consist of only calls that occur at the boundary of an external storage interface; alternately, they may also include synchronization operations internal to the

program itself. Details about internal synchronization may reveal certain dependencies; for example, if two I/O requests at different times were both issued while a given lock was held, we could infer that an ordering in which the two I/O requests are issued concurrently is not a valid point in the I/O space. Internal program logic also affects ordering, however, so tracing locking operations is not a complete solution.

2.2.2 Inference

We now describe three I/O-space inference techniques, including ROOT, based on three different types of trace information. These are summarized in Figure 2.1.

Figure 2.1(a) illustrates a *deductive inference* approach based on active tracing. Active traces allow methodical exploration of the I/O space via controlled experimentation. //Trace is an example of an active-tracing tool [99]. An I/O space can be determined by collecting numerous traces, artificially slowing different requests each time, and observing which other requests are delayed as a result. While this is an elegant approach, it is inconvenient and time consuming to collect many traces, especially at the slowed speed. In production systems, delaying I/O in this manner may be unacceptable, and collecting traces multiple times with the same input may not be possible.



(a) deductive inference

(b) statistical inference

(c) hint-based inference (ROOT)

Figure 2.1: Techniques for I/O-space inference. Active tracing perturbs timing by artificially delaying specific events so as to observe which other events are affected; passive tracing allows all events to occur at their natural pace.

Figure 2.1(b) illustrates a *statistical inference* approach based on passive tracing. Some debugging tools use this approach to infer the causal relations between RPC calls [6], though we are not aware of any trace replay systems that take this approach. This approach has the advantage that traces are much easier to collect and doing so does not artificially degrade performance (beyond the overhead of the tracing itself, that is). However, it is likely that much of the I/O space will not be explored unless traces are collected in many different environments.

Figure 2.1(c) shows the goal of the ROOT approach: to infer as much as possible about an I/O space given a single passively-collected trace with no details about application-internal synchronization details. Inferring anything about an I/O space given a single data point might seem challenging; however, the resource access patterns of even a single trace can provide useful hints about the I/O space. For example, if a program performs two reads from the same file, the reads may use the same file descriptor for both requests, or different file descriptors. The use of different file descriptors may indicate that the reads are unrelated, and hence could be replayed in a different order, or even concurrently.

While a human reading through a trace would likely be able to infer more application-level logic than an automated

tool, creating benchmarks via manual trace inspection would be an unpleasant task. Thus we propose a new approach called ROOT: Resource-Oriented Ordering for Trace replay. ROOT defines a trace model, making it easier to create tools that reason about traces. ROOT also defines a notation for expressing the "hints" a human reading a trace might use to make a reasonable guess about the target program's dependency properties. The details of ROOT are provided in Section 2.3.

The ROOT approach can sometimes make incorrect inferences – its inferences are, ultimately, based only on hints, which can be misinterpreted. We do not attempt to make more accurate inferences than the deductive or statistical methods; those techniques have the advantage of being based on a great deal more data. The ROOT approach is useful when a realistic benchmark is desired, but trace data from the original application is limited. Such cases are common, such as when studying traces of production systems, where inputs may be uncontrollable and the overheads of active tracing are unacceptable. Furthermore, it is already relatively uncommon for companies to collect and share traces; motivating them to collect active traces or enough traces to apply statistical inference may be infeasible.

One weakness of ROOT is that it assumes the I/O space

will consist of different orderings of a single set of I/O actions. Given a series of actions in a trace, it is reasonable to infer how they might be reordered; however, it is essentially impossible to correctly guess that a program sometimes generates a certain request if that request never actually appears in the available trace data. We do not view this limitation as problematic; inference based on methodical exploration could hypothetically deduce I/O spaces consisting of a varying set of actions, but existing tools based on this approach (e.g., //Trace) have the same limitation.

2.3 ROOT: Ordering Heuristics

By enforcing an approximately-correct partial ordering on replay actions, replay tools can generate realistic I/O that resembles the original program's behavior. In this section, we define ROOT's hint-based ordering rules for replay. Our constraints are oriented around resources, such as files, paths, and threads. The key idea is that the set of actions involving a given resource should be replayed in a similar order as in the original trace. If all actions in a trace interact with the same resource, then replay will be highly constrained, but if there is little overlap between the resources touched by different actions, there will be little constraint on the replay order.

Although resource-oriented ordering is simple in theory, real storage systems have complex, many-to-many relationships between actions and resources; some types of actions (e.g., directory renames) can impact an arbitrarily large set of resources (e.g., paths). The relationship between an action and the resources it touches cannot be inferred by looking at the trace record for the action by itself. Rather, inferring the relationships requires a trace model that considers each action in the context of the entire trace and an initial snapshot of system state.

We will now describe a general trace model applicable to traces from a variety of storage systems (e.g., key-value stores or file systems), define and intuitively justify several rules that can be applied to a trace to obtain a partial ordering of actions with which to guide replay, and describe ARTC's use of our trace model and ordering rules to replay system-call traces.

2.3.1 Trace Model

A *trace* contains a totally-ordered series of *actions*. The types of actions are system specific; a key-value store might have `put`, `get`, and `delete` actions, whereas a file system might have `open`s, `read`s, and `write`s. Each action interacts with one or more *resources*; threads, keys, values, paths, and files

are examples of resources.

A simple file rename across directories might involve five resources: the thread performing the rename, source and destination paths, and the directories containing these paths. Conceptually, an *action series* is associated with each resource, consisting of all the actions related to the resource in the order they occurred in the original execution. All our rules are based on action series; it is, however, unnecessary to ever materialize such lists.

Some resources point to other resources. For example, a path might point to a directory, which in turn might point to other paths. Some actions that touch a resource also touch all other resources it transitively points to.

Some resources have *names* that appear in the trace. A file resource does not itself have a name, but it might be pointed to by a path, which does. The same name might apply to different resources at different points in a trace; for example, "3" could be a name designating different file descriptors at different times. Our model differentiates uses of the same name with *generation numbers*, increasing integers associated with each such use, which together with a name uniquely identify a resource.

Figure 2.2 provides an example showing how action series are derived from a system-call trace. The series for thread

(a) Example Trace

```
1 [T1] mkdir("/a/b")                = 0
   Resources:
   T1,dirA,dirB,path(/a/b)
2 [T1] open("/a/b/c",CREATE)        = 3
   T1,dirB,file1,path(/a/b/c),fd3
3 [T1] write(3, ...)                = 8
   T1,file1,fd3
4 [T1] close(3)                     = 0
   T1,file1,fd3
5 [T1] rename("/a/b", "/a/old")     = 0
   T1,dirA,dirB,file1,four paths...
6 [T2] open("/x/y/z")               = 3
   T2,dirY,file2,path(/x/y/z),fd3
7 [T2] open("/a/b")                 = 4
   T2,dirA,file3,path(/a/b),fd4
                  ...
```

(b) Action Series

Resource	Actions
thread(T1)	1,2,3,4,5
thread(T2)	6,7
dirA	1,5,7
dirB	1,2,5
dirY	6
file1	2,3,4
file2	6
file3	7
path(/a/b)@1	1,5
path(/a/b)@2	7
path(/a/b/c)@1	2,5
path(/a/old)@1	5
path(/a/old/c)@1	5
path(/x/y/z)@1	6
fd3@1	2,3,4
fd3@2	6
fd4@1	7

Figure 2.2: Example action series. A snippet from a simple system-call trace for two threads is shown in 2.2(a). Beneath each event, a comment lists the resource touched by each system call. 2.2(b) shows the action series corresponding to each resource that appears in the trace.

`T1` is simply the set of actions executed by the thread (1, 2, 3, 4, 5), in the order they were executed. The series for `dirA` (1, 5, 7) is the set of actions that accessed `dirA`, in the order they occurred. Note that action series do not distinguish between subjects (e.g., threads) and objects (e.g., directories). The figure also shows different action series for `fd3@1` and `fd3@2`. This "name@generation" notation is used to distinguish between resources when the same name is used for different resources at different times. Here, `3` is a shared name for the file descriptors created in actions 2 and 6.

2.3.2 Ordering Rules

Section 2.2.2 suggested that how a program manages resources, as shown in a trace, provides hints about its I/O space. Given a trace model, we can now discuss these hints more formally and define our replay rules.

The rules we define determine an I/O space for a replay benchmark. Ideally, the I/O space for the benchmark will be similar to that of the original application. However, there are two ways in which we might deviate from this goal.

First, a rule might be excessively restrictive, resulting in *overconstraint*. In this case, the replay's I/O space omits points that would be present in the I/O space of the original program. For example, a hypothetical rule that (perhaps in

a heavy-handed attempt to prevent runtime errors during replay) serialized all file creation and deletion operations would necessarily preclude any replay ordering involving any concurrency between multiple operations of this type, even if the original program might happily perform them that way.

Second, a rule might be insufficiently restrictive, resulting in *underconstraint*. In this case, the replay I/O space may contain an ordering for an I/O set that the original I/O space does not contain. Underconstraint could arise if, for example, a replay did not enforce the ordering requirement that a read from a file descriptor occurs only after that file descriptor has been opened, leading to multiple forms of potential runtime misbehavior (the read either failing with an `EBADF` error or successfully reading data from a different open file that happened to share the same file descriptor number).

We say that a *stronger* rule A *subsumes* a *weaker* rule B if the orderings allowed by rule A are a strict subset of those allowed by rule B. In this case, if B causes overconstraint, A will as well. Likewise, if A allows underconstraint, B will as well.

We have identified three rules based on action series that are useful for replay; these are summarized in Figure 2.3. The first rule, *stage ordering*, simply says that an action that creates a resource must be played before any uses of the

Rule	**Definition**
Stage	$acts$[create] < $acts$[i] < $acts$[delete]
Sequential	$acts$[i] < $acts$[i+1]
Name	N@G.$acts$[last] < N@(G+1).$acts$[first]

Figure 2.3: Ordering Rules. *a1* < *a2* means action *a1* must be replayed before action *a2*. $acts$[create] and $acts$[delete] represent $acts$[first] and $acts$[last], respectively, when the first action in a series is a create or when the last action is a delete. When this is not the case, the constraint does not apply.

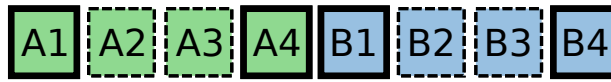
resource, and also that any uses of the resource must be played before a deletion. The intuition behind stage ordering is that when we observe a successful event in a trace, we assume the program took some action to ensure success, so replay should do likewise.

The second rule, *sequential ordering*, forces all actions involving a resource to replay in the same order as in the original trace. Sequential ordering is a stronger constraint, subsuming stage ordering, but may lead to overconstraint. For example, if multiple reads from the same file all touch the same resource, it may in fact be correct to allow these reads to be reordered during replay, but sequential ordering would disallow this. In contrast, stage ordering might be too weak: reordering two reads from the same file could be incorrect if

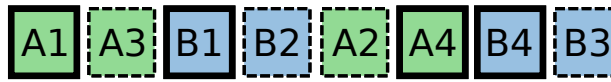
the first retrieves indexing information and the second relies on the result of the first to determine where in the file to read from. The intuition behind sequential ordering is that data dependencies may be more likely when actions access the same resources rather than disjoint sets of resources; constraints should be tighter in such cases.

The third rule, *name ordering*, requires that the action series of different generations of the same name are neither overlapped nor reordered during replay. Sequential- and name-ordering each allow some orderings not allowed by the other. The intuition behind name ordering is that when a programmer reuses the same name for different resources, the resources are likely related.

Figure 2.4(a) shows an example trace of actions on two resources, A and B, that use the same name at different times. Figure 2.4(b) gives an example replay ordering, and Figure 2.4(c) describes how the replay would violate different ROOT rules. The replay of generation A is allowed by stage ordering because the sequence begins and ends with create and delete actions, respectively, but violates sequential ordering because the two middle actions (A2 and A3) are reordered. The replay of generation B violates stage ordering because the deletion action is not last, and thus also violates sequential ordering. Finally, actions belonging to generation B start



(a) Original trace order



(b) Replay order

	Stage	Sequential	Name
Generation A	*none*	A3 < A2	B1 < A4
Generation B	B4 < B3	B4 < B3	

(c) Violations

Figure 2.4: Examples of valid and invalid orderings. Each square represents an action. Different colors represent consecutive generations of the same name. Thick borders indicate creation and deletion events.

replaying before A is finished, which violates name ordering since A and B are different generations of the same name.

Because rules vary in strength, one must decide which rules to apply to which resources when employing ROOT. In 2.4.2, we describe ARTC's default use of the rules for Unix filesystem resources and the reasoning for each. More broadly, however, we suggest three guidelines for applying the

rules in a new context. First, domain knowledge should be used. For example, if it is known that a programmer generally intentionally chooses names for a certain resource (e.g., a path name), name ordering should apply, but if the names are chosen arbitrarily, name ordering might cause overconstraint. Second, the costs of different types of mistakes should be taken into account; overconstraining a replay might skew the timings of certain actions, but underconstraining might cause the actions to fail, and thus finish instantly. Third, if many actions fail during replay, underconstraint is a likely cause.

2.4 ARTC: System-Call Replay

We now describe ARTC, a benchmarking tool that applies the ROOT approach to system-call trace replay on Unix file systems. We now discuss goals for the tool (2.4.1), demonstrate how the three ROOT rules abstractly defined in the previous section concretely apply to Unix file systems (2.4.2), and detail our implementation (2.4.3).

2.4.1 Goals

The aim of ARTC is to be a broadly applicable storage benchmarking tool, offering a flexible set of parameters while remaining easy to use.

Portability: ARTC attempts to support realistic cross-platform replay. Because traces from one system often include system calls that are not supported on others, ARTC emulates these calls, issuing the most similar call (or combination of calls) on the target system.

Ease of use: ARTC benchmarks make it simple for end users to apply them to a file system. All that is required for basic use is the compiled benchmark and a directory in which to run the benchmark (perhaps the mountpoint of a file system to be evaluated). There is no need to describe a benchmark using a specialized configuration language or determine the values of non-default parameters to measure the performance of a file system. Also, ARTC makes it easy to create new benchmarks by supporting standard tracing tools that are often pre-installed in Unix environments (e.g., `strace`).

Flexibility: ARTC provides a variety of optional tuning parameters, controlling how initialization is done, the speed at which actions are replayed, the ability to disable specific ordering constraints, and how certain actions are emulated during cross-platform replay.

Correctness: ARTC attempts to generate benchmarks with nondeterministic behaviors resembling the nondeterminism of the original applications as closely as possible given the

Resource	Stage	Sequential	Name
program		●	
thread		$\bullet_{\text{req}}$	
file	○	●	
path	$\bullet_{\text{joint}}$	○	$\bullet_{\text{joint}}$
fd	●	●	
aiocb	●	○	○

Figure 2.5: Replay modes. Circles represent reasonable ways to apply rules to resources; filled circles are modes currently supported by ARTC. `thread_seq` is always required; `path_stage` and `path_name` must be applied jointly. All supported rules except `program_seq` are enforced by default.

information available in the traces. Despite this nondeterminism, ARTC's ordering constraints enforce that the replay's *semantics* should match those of the original trace as closely as possible.

2.4.2 ROOT with System-Call Traces

We now discuss the application of ROOT to system-call traces. We consider six types of resources: programs, threads, files, paths, file descriptors (FDs), and asynchronous I/O control blocks (AIOCBs). We focus on single-process replay, so all the actions in a trace are associated with a single program resource, as well as one of the many thread resources. Many

actions will access file resources via paths and file-descriptor resources. Finally, AIOCBs are used to manage asynchronous I/O on file descriptors; AIOCBs point to file descriptors.

Figure 2.5 shows which rules could reasonably be applied to which resources and which are supported by ARTC's replay modes. Though all supported constraints except `program_seq` are enforced by default, ARTC allows any combination of ordering modes to be selected for replay, with two restrictions. First, sequential ordering is always applied to threads; second, for paths, stage and name ordering may only be applied jointly. A discussion of the replay modes follows:

Programs: All actions in a trace involve a single *program* resource. Applying sequential ordering to the program represents the `program_seq` replay mode. `program_seq` is ARTC's strongest replay mode, subsuming all other modes; however, `program_seq` forces a total ordering on replay, typically resulting in severe overconstraint (the performance impact of `program_seq` is demonstrated in 2.5). Stage ordering does not make sense for the program resource because no action in the trace can be said to "create" the program; name ordering is irrelevant as there are not multiple generations of program resources in a single trace.

Threads: Each action in a trace is performed by exactly one *thread* resource. ARTC always enforces `thread_seq`

mode, as it has no simple way to reorder actions within a thread during replay. In general, the order of actions performed by a single thread provides a good hint about program structure. Some patterns, however, such as thread pools, are clear exceptions; ARTC cannot infer these types of program structures. However, we are not aware of any other replay tools that can do so without additional details about program internals. Stage and name ordering do not apply to threads for the same reasons they do not apply to programs.

Files: We define a *file* as the data associated with a specific piece of metadata, such as an inode number. Inode numbers, however, do not appear in our traces, so the existence of files is only implicit. An accurate filesystem model that considers symbolic links, hard links, and the behavior of various system calls allows us to determine when different paths (or file descriptors) refer to the same file, as well as when the same path name refers to different files at different times. Because files do not appear explicitly in traces, name ordering is irrelevant. Stage and sequential ordering apply, though; ARTC supports the latter with `file_seq`, a fairly strongly-constrained replay mode. When other resources refer to files, as they often do, `file_seq` subsumes stage or sequential ordering when applied to those resources. However, the rules for the following resources do prevent some orderings

`file_seq` allows, such as when name ordering is relevant or when the resources refer to directories rather than regular files.

Paths: *Path* resources point to file resources and have names that appear in traces. All our ordering rules could be applied to paths; ARTC supports the joint application of stage and name ordering with `path_stage+` mode. We do not support stage ordering without name ordering; doing so would require the use of substitute names during replay. For example, if a trace shows that a path `"foo"` referred to different files at different times, replay would have to either prevent concurrent access to those files during replay (*i.e.*, use name ordering), or use substitute names (e.g., `"foo1"` and `"foo2"`).

Applying stage ordering to paths assumes that when a trace action makes a successful access to a path, the program must have taken some measure to ensure its success. We believe this is a good hint in general, but it may sometimes cause overconstraint. For example, programs may use the `stat` call (which fails when a path does not exist) to determine whether a path exists. If a `stat` call succeeds during the original execution, it may be a coincidence; during replay, if certain actions finish sooner than they did during trace collection, it may be correct to replay a `stat` call sooner, even

if the call would fail.

Similarly, applying name ordering assumes that different files are related if they use the same path name at different times. Because programmers or users choose most path names, we believe this to be a meaningful hint. While this is usually the case, one common exception is when path names are chosen arbitrarily (e.g., names for temporary files). In this case, `path_stage+` may lead to overconstraint, but we suspect this situation is rare in practice since random file names are not generally chosen from a small set of possibilities and hence are unlikely to collide with each other.

File descriptors: Successfully opening a path produces a file descriptor (FD), which acts as another type of pointer to a file. ARTC supports stage ordering (`fd_stage` mode) and sequential ordering (`fd_seq` mode) for FDs. Although FDs have integer names that appear in a trace, these names are usually chosen by the operating system, so they provide no hints about the I/O space; thus, name ordering is of no real use for FDs. Additionally, since FD names are small integers, they can be easily remapped using a simple array, allowing descriptors that used the same name in the original trace to coexist simultaneously during replay.

Asynchronous I/O control blocks: Asynchronous I/O may be performed by wrapping a file descriptor in an asyn-

chronous I/O control block (AIOCB) structure and submitting it in a request to the file system. Because file descriptors point directly to files, AIOCBs point indirectly to files. ARTC supports stage ordering for AIOCBs with `aio_stage` mode. Applying sequential ordering could also be potentially useful, even though ARTC does not currently support it.

2.4.3 Implementation

Figure 2.6 show an overview of the main components of ARTC. Given a system-call trace and an initial file-tree snapshot collected on a source system, the ARTC compiler automatically generates a benchmark. The ARTC replayer uses the file-tree snapshot to initialize on the target machine an equivalent filesystem tree in which the actions in the trace are replayed. Filesystem APIs vary slightly across systems, so ARTC emulates recorded actions via the closest equivalent on the target machine when necessary, supporting replay on Linux, Mac OS X, FreeBSD, and Illumos.

ARTC's implementation consists of approximately 12,000 lines of C and 4,000 lines of `bison` and `flex` grammars (as measured by `wc -l`), and is capable of replaying over 80 different system calls. A significant portion of the code is shared between the ARTC compiler and the ARTC replayer, but the two components comprise roughly equal fractions of

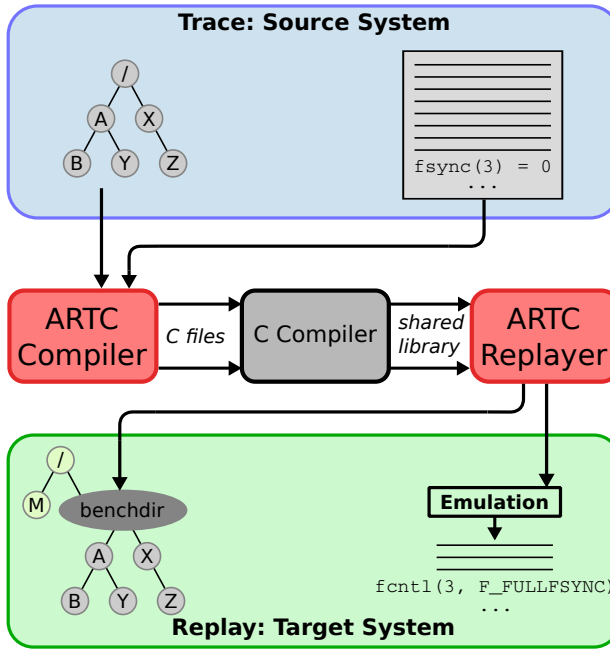


Figure 2.6: ARTC Components. From the source system we collect an initial snapshot of filesystem state and a trace of application system calls. The ARTC compiler translates these into C code representing a set of static data structures that are compiled into a shared library. The ARTC replayer then loads this library and uses the data inside it to initialize the filesystem and replay the trace on the target system.

the code size.

Compilation

ARTC currently supports `strace` output and a special `dtrace`-generated format used by the iBench traces (see 2.5.1), but trace parsing is cleanly separated from the core processing functionality, so ARTC can be readily extended to support new input formats. However, the core functionality assumes the following information will be available for each system call in the trace:

- Entry/return timestamps
- Numeric ID of issuing thread
- Type of call (e.g., `open`, `read`, *etc.*)
- Parameters passed
- Return value

Some system-call parameters are not actually required; for example, ARTC ignores the buffer pointers passed to `read`. While our trace model could theoretically treat buffer pointers as another type of resource, we suspect buffer reuse would make it impossible to derive meaningful hints from the additional information.

In addition to a trace of actions, ARTC requires an initial snapshot of the parts of the filesystem tree that the program accesses. It is unnecessary to record actual file contents in the snapshot; however, it is important to record the contents of directories, the sizes of files, and references made by symbolic links. Having an accurate model for symbolic links is crucial to enforcing the `file_seq` rule. Even when the same file is accessed via different paths, `file_seq` must constrain the accesses to be replayed in the same order as in the trace.

Given a trace and an initial snapshot, ARTC automatically generates C code, which is then compiled into a shared library. The shared library is later loaded by a general tool for replay (2.4.3). The generated code consists of tables of static data (arrays of `struct`s) describing the resources and actions in the trace. We chose to generate C code as a simple way to serialize the replay information; generating input files that the replay program parses would work as well, though using pre-built data structures saves the runtime overhead of parsing a more generic input format.

Initialization

Before replay, it is necessary to restore the initial state snapshot in the directory where the benchmark will execute. During this stage, ARTC creates the necessary directories, popu-

lating them with files of the appropriate size containing arbitrary data, and creates any necessary symbolic links. Some special files (such as `/dev/random`) are created as symlinks to the corresponding special files in the target's root filesystem.

Because initialization may take much longer than the actual replay of some traces, ARTC can perform a *delta init* that is useful when most of the init files are already in place (e.g., the file tree was previously initialized, and a prior replay only slightly modified the tree). Delta init only creates, deletes, or changes of the sizes of existing files as necessary to restore the initial state.

Initialization is not a major focus of our work, but ARTC could be extended to use initial snapshots with richer information about invisible filesystem state. For example, for a log-based file system, replay speed will depend greatly on the order in which the initial files are created. A more sophisticated initialization could account for this, and even reproduce the fragmentation that occurs due to aging in real-world deployments [5, 129].

ARTC also includes options that make it easy to initialize overlaid filesystem trees based on the snapshots for multiple traces, so that multiple traces can be replayed concurrently. For example, one could use Magritte (2.6), our benchmark suite of Apple desktop applications, to run a workload similar

to a user browsing photos in iPhoto while listening to music in iTunes.

Replay

ARTC's replayer is the component that actually performs system-call replay, enforcing the enabled ordering modes while doing so. Although our discussion of ordering modes has been in terms of action series, ARTC, like the programs that generate the traces to begin with, does not need to explicitly materialize such lists. Rather, ARTC enforces rules using standard synchronization primitives and the dependency information determined by the compiler. Each system call (action) includes a condition variable that other threads can wait on if an action they are about to replay is dependent on that action. For example, before a given thread replays an action that uses a certain file descriptor, it checks if the `open` call that created that file descriptor has already been replayed, and if not, waits on the `open` action's condition variable. When the replay of an action completes, the thread that replayed it performs a broadcast operation on the action's condition variable in order to wake any threads that may be waiting on it.

Stage ordering: except for a resource's *create* action, all other actions will wait on the create action before replaying,

enforcing that it is the first of that resource's associated actions to replay. *Delete* actions have a dependency on each other use of the resource, though for space-efficiency reasons our current implementation uses a separate structure for the resource with a count of remaining uses and a condition variable of its own.

Sequential ordering: Each action belongs to the action series of one or more events. For each such series, the action in question has a dependency on the previous action in the series, and correspondingly waits for its completion before proceeding with its own replay.

Name ordering: When an action is the first of a new generation of a resource on which name ordering is applied, it has a dependency on the last event of the preceding generation, and waits for it to complete.

We use this resource and action bookkeeping to enforce all ordering rules except `thread_seq` and `program_seq`. Because sequential ordering is always enabled for threads, we simply use a replay thread for every thread that appeared in the original trace. Each of these threads loops over its own actions from the original trace, playing each one in order once all its dependencies are satisfied. When `program_seq` is used, all trace actions are instead replayed from a single replay thread in the order in which they appeared in the original trace.

Besides enforcing ordering rules during replay, ARTC is

also capable of considering timings from the original trace. For example, the original trace might show that even after all the inferred dependencies for an action are satisfied, the action is executed after some time interval, which we call *predelay*. Predelay may be due to computation. It is not our goal to have a sophisticated model of computation, but ARTC provides some basic options for incorporating predelay during replay. ARTC may ignore predelay (AFAP, or as-fast-as-possible mode), sleep for the predelay time (natural-speed mode), or use some multiple of predelay, perhaps based on CPU utilization information (if available). Given our simplistic model of computation, we do not expect ARTC to produce accurate timings for compute-bound workloads.

After finishing replay of the entire trace, the replayer outputs basic timing information, such as the elapsed wall-clock time, as well as detailed data about why a replay performed the way it did, such as per-thread timing reports and latencies for each call. Additionally, details about the similarity of system-call return values during replay to return values during trace collection are generated (*i.e.*, the semantic accuracy of the replay), providing indications of possible underconstraint.

Emulation

Supporting cross-platform replay is challenging, as each Unix-like platform has its own slightly distinct API for filesystem access. For such system calls, there are usually near equivalents on other platforms, but occasionally a call provides a unique primitive. In order to support such calls, ARTC converts them to pseudo-calls. During replay, ARTC emulates pseudo-calls by using the most similar system calls available, sometime executing multiple calls on the target system to emulate a single call on the source system.

ARTC performs emulation for 19 different calls. 11 of these cases are for special metadata-access APIs (e.g., extended attributes); not only do the names of the calls differ in these cases, but some systems support parameters and options not supported by others. When emulating these calls, we simply ignore such parameters.

Another three cases pertain to filesystem hints; in particular, prefetching, caching, and preallocation hints are all treated slightly differently on each platform. Linux, Mac OS X and Illumos generally offer equivalent functionality, though sometimes via different APIs; emulation for these is straightforward. On FreeBSD, however, we simply ignore some of these calls where analogous APIs are not available. Three more emulations are required for obscure, undocumented Mac

OS X system calls, that appear to be metadata related and are hence emulated with small metadata accesses.

Another case addresses a difference in `fsync` semantics on different systems. Linux filesystems typically force data to persistent storage when `fsync` is called, but on Mac OS X semantics are different, and data is merely flushed to the device, which may cache it in volatile memory; `fcntl(F_FULLFSYNC)` is necessary to achieve true durability. When replaying traces collected from Linux on a Mac, a replay option determines which semantics are used to emulate `fsync`.

The final case is the `exchangedata` call, a unique atomicity primitive provided by Mac OS X. Given two files, `exchangedata` performs an atomic swap such that each file's inode points to the other file's data, preserving inode numbers and other metadata. Although there is no truly atomic equivalent on other platforms, we emulate this via a `link` and two `rename`s.

2.5 Evaluation

We evaluate ARTC by establishing its preservation of semantic correctness and comparing its performance accuracy with a set of simpler strategies.

The simplest approach we compare against is *single-threaded* replay, which issues all calls in the trace from a single replay

thread in the same order in which they were issued in the trace. This approach precludes not only reordering but also any concurrency between system calls. *Temporally-ordered* replay also issues calls during replay in the order they were issued during tracing, but uses one replay thread per traced thread, so calls that overlapped during tracing may be issued concurrently during replay. While it permits some concurrency, this approach allows no real reordering to occur during replay. *Unconstrained* replay falls at the opposite end of the ordering spectrum, employing multiple threads but enforcing no synchronization between them. This approach allows maximal reordering (within the constraints of `thread_seq`, which is still implicitly enforced) but is vulnerable to race conditions involving shared resources.

All of these replay strategies are actually implemented as alternative modes of operation of ARTC's replayer. Various command-line flags can be specified to disable multithreading (for single-threaded replay), enable enforcement of the `program_seq` rule (providing temporally-ordered replay), or disable enforcement of all rules (providing unconstrained replay). References to ARTC replay in the remainder of this section refer specifically to ARTC running in its *default* mode of operation (multiple threads, with all rules except `program_seq` enforced).

	Trace	Failed Events UC	TO	ST	ARTC	Total Events
iMovie	add1	51	3	3	3	24,655
	export1	4,538	5	5	5	42,697
	import1	4,437	7	7	7	35,733
	start1	43	2	2	2	21,375
iPhoto	delete400	298	2	2	2	472,393
	duplicate400	53,226	2	2	2	210,612
	edit400	881,714	2	2	2	1,660,736
	import400	377,873	3	3	7	827,964
	start400	74	2	2	2	35,547
	view400	76,375	2	2	2	278,217
iTunes	album1	549	0	0	0	9,671
	importmovie1	56	0	0	0	5,290
	importsmall1	1,459	0	0	0	10,739
	movie1	2,578	0	0	0	9,507
	startsmall1	3	0	0	0	5,466
Keynote	create20	269	0	0	0	36,434
	createphoto20	733	2	2	2	38,549
	play20	0	0	0	0	28,822
	playphoto20	208	0	0	0	30,055
	ppt20	4	0	0	0	51,620
	pptphoto20	4	0	0	0	126,506
	start20	0	0	0	0	17,775
Numbers	createcol5	59	0	0	0	15,069
	open5	0	0	0	0	12,028
	start5	0	0	0	0	10,067
	xls5	0	0	0	0	14,544
Pages	create15	36	4	4	4	16,520
	createphoto15	401	4	4	4	56,024
	doc15	4	4	4	4	15,696
	docphoto15	139	4	4	4	205,566
	open15	4	4	4	4	15,091
	pdf15	4	4	4	4	15,213
	pdfphoto15	106	4	4	4	54,488
	start15	4	4	4	4	13,927

Figure 2.7: Replay failure rates. The number of event-replay failures in each trace is shown for a completely unconstrained multithreaded replay (UC), temporally-ordered replay (TO), single-threaded replay (ST), and ARTC, all in AFAP mode. Each data point is the largest failure count observed in five runs. The rightmost column shows the total number of replayed actions in the trace.

2.5.1 Semantic Correctness: Magritte

We evaluate the semantic correctness of ARTC's replay by examining its behavior with 34 traces of Apple's iLife and iWork desktop application suites [56]. The complex inter-thread dependencies and frequent metadata accesses found in these traces make them an excellent correctness stress test. We also believe these traces are useful beyond this evaluation, and so we release the compiled traces as a new benchmarking suite called Magritte.[1] Before presenting the results, we describe some of the difficulties we encountered in the process of replaying these traces:

Special files: Some of the traces include reads from `/dev/random`, which resulted in extremely slow reads on Linux (tens of seconds for less than a hundred bytes of data). On Mac OS X, `/dev/random` is a non-blocking source of random bytes, whereas on Linux, reads from `/dev/random` block when the kernel judges that its entropy pool is depleted. We solve this by creating `/dev/random` as a symlink to `/dev/urandom`, which does not block, when replaying on Linux.

External bugs: We encountered some behaviors on Mac OS X that appear to simply be kernel bugs. Calling `close`

[1] Magritte is named for the 20th-century Belgian artist René Magritte, who created a number of paintings prominently featuring apples, most notably *The Son of Man* in 1964.

on a file descriptor returned from `shm_open`, for example, consistently reports failure with `EINVAL`, which is not listed in its documentation. Interestingly, the call appears to succeed, since subsequent `open`s then return file descriptors re-using the same value. ARTC generally outputs warnings when replayed calls do not conform to its expectations, but sometimes suppresses them in cases such as this.

Missing trace details: There are a handful of sequences in the traces for iTunes that show system calls of the form `open(path, O_CREAT|O_EXCL)` executing successfully, but at points where prior events in the trace would indicate that `path` should already exist. While we cannot be entirely sure of the cause of this, it may be due to a mistake in the collection of the traces from the original applications. ARTC handles these by simply replaying them without the `O_EXCL` flag.

After addressing these issues, we replayed the traces with each of the four modes. In order to amplify concurrency and best exercise each mode's enforcement of the trace's semantics, we performed these replays in AFAP mode on an SSD-backed ext4 file system, and did not clear the system page cache between each benchmark's initialization and execution. Figure 2.7 shows the number of errors in trace replay for each replay mode; with the exception of `iphoto_edit400`, the failure counts for single-threaded and temporally-ordered

modes are identical to those of ARTC on all traces. Each reported error count is the largest number of errors observed over five replays of the trace.

Although unconstrained replay is semantically correct when replaying some traces (e.g., `keynote_start20`), many replays produce thousands of errors; on `iphoto_edit400` over half the trace's events replay incorrectly. Not only are the failure rates for ARTC and the other highly constrained modes several orders of magnitude lower, further investigation reveals that almost none of ARTC's errors are due to invalid reordering. Rather, except for four failures in `iphoto_import400`, all of ARTC's failures are due to a lack of extended attribute initialization information in the iBench traces; replay initialization thus does not create these attributes, and replayed calls attempting to access them fail. The four failures caused by reordering in `iphoto_import400` are due to an edge case involving a directory rename un-breaking a broken symlink, which ARTC's filesystem model does not currently handle, causing it to miss some path dependencies and thus allow some invalid reorderings.

Given the unconstrained mode's extreme error rate, we do not consider it a viable option, and thus do not consider it in the remainder of our evaluation. We do not use Magritte for the performance accuracy aspect of our evaluation because

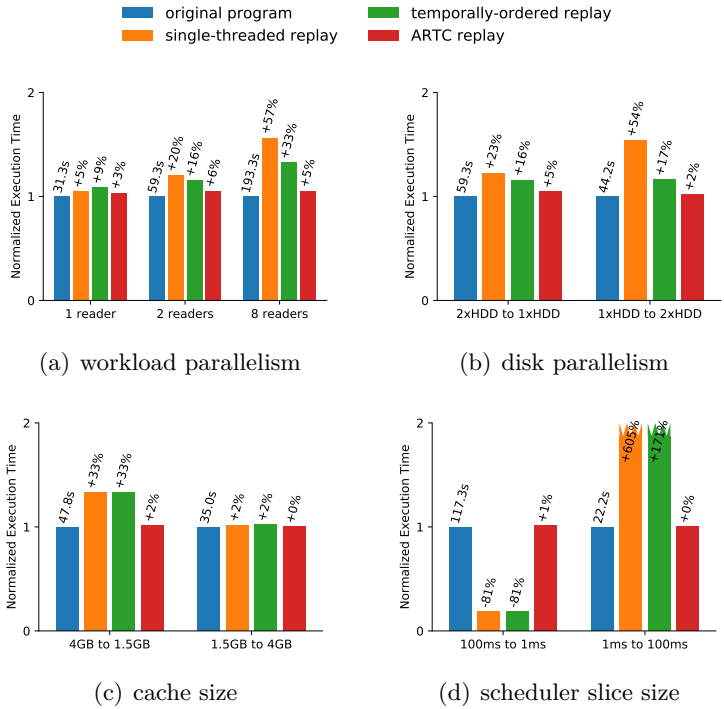
the workloads are interactive and thus not consistently I/O-bound, an operating mode ARTC does not focus on modeling accurately.

2.5.2 Performance Accuracy

Here we employ micro- and macro-benchmarks to evaluate ARTC's performance accuracy, which we find is substantially better than that of the simpler single-threaded and temporally-ordered replay methods.

Microbenchmarks

In this section, we use microbenchmarks to explore feedback effects between workloads and storage systems, showing how each naturally affects the other. In one experiment, we adjust the degree of parallelism in the workload and show how the storage system takes advantage of the additional flexibility offered by increased queue depths. In three further experiments we construct feedback loops, changing aspects of the storage system in ways that should change the workload's behavior. We experiment with varying disk parallelism, cache size, and I/O scheduler slice size. We show that in each of these scenarios ARTC adapts in a natural way, but the simpler single-threaded and temporally-ordered replay methods do not.



(a) workload parallelism

(b) disk parallelism

(c) cache size

(d) scheduler slice size

Figure 2.8: Microbenchmarks. Effect of feedback loops on accuracy. Labels on the original-program bars indicate running times for the original program on the target system. Labels on other bars indicate a percentage error relative to the original.

Workload parallelism: For our first experiment, we wrote a simple program the spawns a variable number of threads, each of which reads 1000 randomly selected 4KB blocks from its own 1GB file. We ran and traced the program with 1, 2, and 8 threads. We then performed single-threaded, temporally-ordered, and ARTC replays of each trace. The timing results for the three traces are indicated by the three groups of bars in Figure 2.8(a). Within each group, the first bar indicates the time it takes the original program to run, and the next three bars indicate how long each of the replay methods take. If replay is accurate, the bars in each group will be similar in size to the first bar of the group.

Figure 2.8(a) shows that going from 1 to 2 readers increases execution time from 31.3s to 59.3s, slightly less than double. Going from 1 reader to 8 performs 8× as much I/O, but execution time increases only 6.2×, to 193.3s. The sub-linear slowdown is due to the increased I/O queue depths of the more parallel workload giving the I/O scheduler and disk more freedom to optimize access patterns, increasing average throughput. These optimizations change the order in which I/O requests complete, which in turn affects the subsequent pattern of requests issued by the program. ARTC's replay adapts to these optimizations similarly, and thus achieves a mere 5% error in elapsed time on the 8-thread workload. The

simpler replay methods, however, are not so flexible, and thus overestimate elapsed time by 57% and 33%.

Disk parallelism: Here we compare accuracy when tracing on a single-disk source and replaying on a two-disk RAID 0 target with a 512KB chunk size (and *vice versa*). We use the same simple program as above, running with two threads. Figure 2.8(b) shows ARTC is accurate moving in either direction (2-5% error), and temporal ordering achieves accuracy similar to the 2-thread case of Figure 2.8(a), but single-threaded replay does significantly worse when replaying the single-disk trace on the RAID, as its serial nature renders it incapable of exploiting the array's increased I/O parallelism.

Cache size: The program for this experiment has two threads and is similar to the previously used program with one difference: Thread 1 sequentially reads its entire file before entering the random-read loop. For both tracing and replay, we use a two-disk RAID 0 and 4GB of memory. To limit the cache size during tracing and replay, we run a utility that simply pins 2.5GB of its address space in RAM, leaving only 1.5GB for the cache and other OS needs. The results of tracing with a normal cache and replaying with a small cache (and *vice versa*) are shown in Figure 2.8(c). ARTC is accurate for both source/target combinations, but the simpler methods are accurate only for replay on the 4GB target, producing

timings that are 33% too long for the 1.5GB target.

In the trace collected on the 4GB system, Thread 1's random reads are all cache hits, and thus all finish long before the vast majority of Thread 2's reads are issued. On a target with a 1.5GB cache, most of Thread 1's reads become cache misses, but the simple replay methods wait for Thread 1 to finish before issuing most of Thread 2's requests; this prevents the system from taking advantage of the RAID array's I/O parallelism. In the other direction (1.5GB source to 4GB target), the simple replay methods are accurate. This asymmetry arises because when replaying the 1.5GB source system's trace on the 4GB target, all of Thread 1's random reads are cache hits, so playing them at the wrong time does not degrade performance.

Scheduler slice size: Here we tune Linux's Completely Fair Queuing (CFQ) I/O scheduler to explore a tradeoff between efficiency and fairness. The CFQ scheduler implements anticipation [69] by giving threads slices of time during which requests are serviced. A large slice means the scheduler will attempt to increase throughput by servicing many requests from the same thread before switching to a different thread, at the cost of increasing the latencies seen by other threads. The length of these slices can be adjusted by tuning the scheduler's `slice_sync` parameter; we experiment with values of 1ms

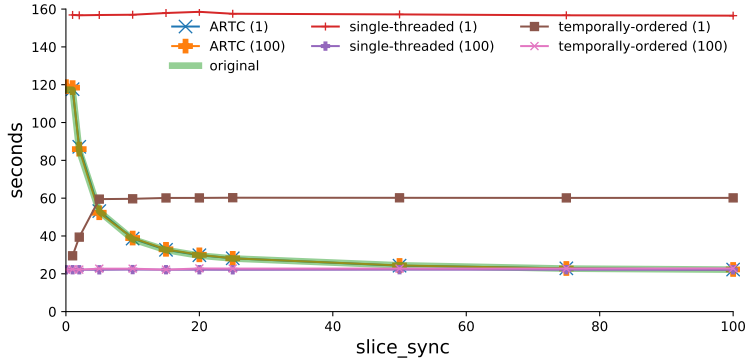


Figure 2.9: Varying anticipation. Throughput achieved by executions with varying `slice_sync` values. Performance is shown for the original program and three replays of two traces (source `slice_sync` values of 1ms and 100ms).

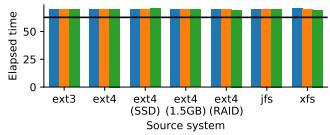
and 100ms. In our microbenchmark program, two threads compete for I/O throughput, each performing sequential 4KB reads from separate large files. Figure 2.8(d) shows that both simple replays dramatically overestimate performance when decreasing `slice_sync` from 100ms to 1ms, and even more drastically underestimate it when moving in the opposite direction. ARTC, however, is extremely accurate in both scenarios.

Figure 2.9 shows the inaccuracy of the simpler replays in greater detail, comparing the original program's performance to each of the three replays on both 100ms and 1ms traces.

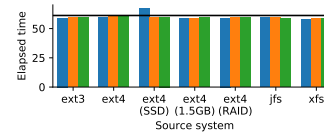
While ARTC predicts the performance of the target system flawlessly, the simple replay methods tend to predict timings that reflect the performance of the source system rather than that of the target. When a trace is collected with a large `slice_sync`, it will show long periods of time servicing requests from a single thread. During replay, even with a smaller slice, a simple replay method will only submit requests from the thread that dominated that period; this effectively reproduces the source system's scheduling decisions at the application level on the target.

Macrobenchmarks

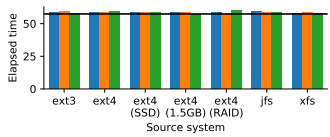
In this section, we stress ARTC's ability to make accurate timing predictions by tracing and replaying the file I/O of LevelDB, an embedded key-value database employed in storage systems such as Ceph and Riak [17, 68]. We evaluated 49 different source/target combinations, exploring various file systems (ext3, ext4, jfs, xfs) and hardware configurations. For each combination, we compare ARTC against single-threaded and temporally-ordered replay, as in Section 2.5.2. We run two benchmark workloads distributed with LevelDB, `fillsync` and `readrandom`, each with 8 threads; `fillsync` threads insert records into an empty database, and `readrandom` threads randomly read keys from a pre-populated database.



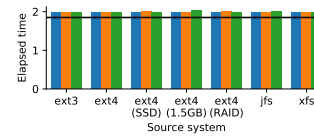
(a) Replays on ext3



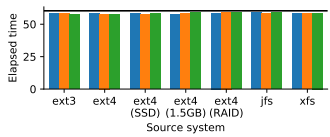
(b) Replays on ext4



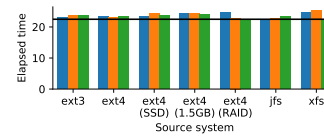
(c) Replays on ext4 (dual-HDD RAID-0)



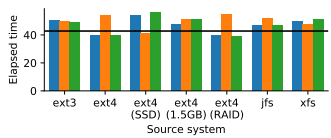
(d) Replays on ext4 (SSD)

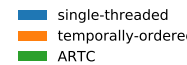


(e) Replays on ext4 (1.5GB cache)



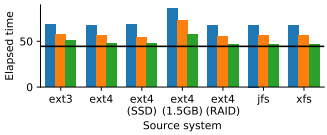
(f) Replays on jfs



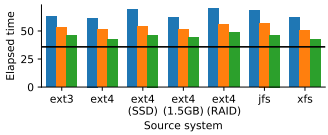


(g) Replays on xfs

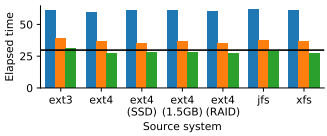
Figure 2.10: LevelDB `fillsync` replays. On each plot, a baseline shows how long the original program runs on the target platform. Bars near this line indicate an accurate replay.



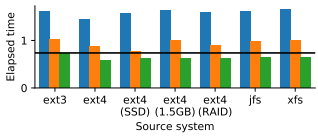
(a) Replays on ext3



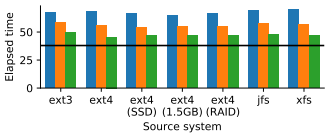
(b) Replays on ext4



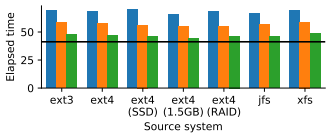
(c) Replays on ext4 (dual-HDD RAID-0)



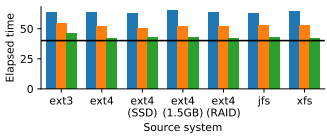
(d) Replays on ext4 (SSD)

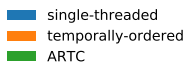


(e) Replays on ext4 (1.5GB cache)



(f) Replays on jfs





(g) Replays on xfs

Figure 2.11: LevelDB `readrandom` replays. On each plot, a baseline shows how long the original program runs on the target platform. Bars near this line indicate an accurate replay.

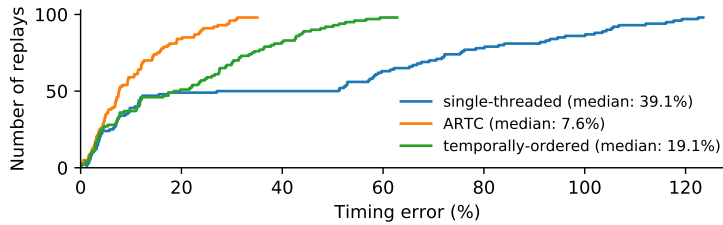


Figure 2.12: LevelDB timing error distribution. This figure shows the distribution of timing errors for the 98 replays performed in each mode.

Figures 2.10 and 2.11 show performance accuracy results for each source/target combination on the `fillsync` and `readrandom` workloads, respectively. The default hardware configuration used a 4GB cache size and a single HDD, though some system configurations used different parameters (SSD or dual-HDD RAID-0 instead of a single HDD, 1.5GB cache size instead of 4GB) where noted in the figures.

For `fillsync`, results are largely uniform (and accurate) across replay modes on all source/target combinations, though replays on xfs do exhibit a slightly greater degree of variation. When multiple LevelDB threads want to issue writes, all writes are issued by one thread; the others simply hand off their data to it. The resulting I/O pattern is essentially that of a simple single-threaded write workload, so simple replay

methods are not at a disadvantage. For `readrandom`, however, both simple methods significantly overestimate execution time in every case. ARTC sometimes overestimates and sometimes underestimates, but its errors tend to be much smaller.

Figure 2.12 shows the distribution of timing errors across all replays. ARTC does best at avoiding extreme inaccuracy; among the least accurate 10% of each method's replays, ARTC averages 28.7% error, compared to 52.9% for temporal ordering and 113.3% for single-threaded replay. Across all replays, temporal ordering and single-threaded replays achieve mean timing errors of 21.3% and 43.5%, respectively, whereas ARTC's replays average within 10.6% of the original program's execution time.

Simple replay methods overestimate `readrandom`'s execution time due to a lack of ordering flexibility, as shown in Figure 2.13, a dependency graph of a representative period of time in a trace of a 4-thread LevelDB `readrandom` workload. Note that there are many more ARTC resource-dependency edges than are shown in this subgraph; however, these edges tend to be between nodes (system calls) that are separated by a long period of time and thus do not fit in the window of time shown here (only edges whose endpoints are both within that span of time are included). Over the entire trace, there are 9135 temporal-ordering edges and 6408 ARTC edges. How-

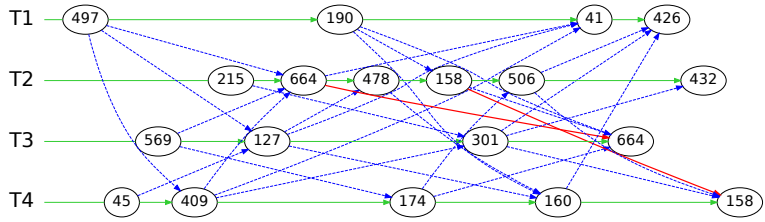
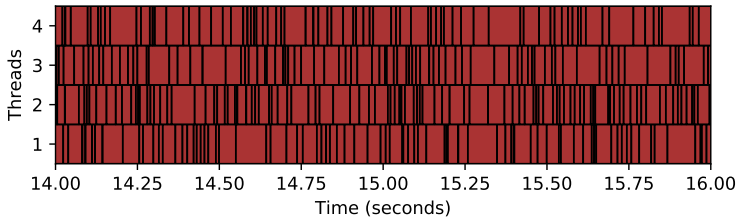


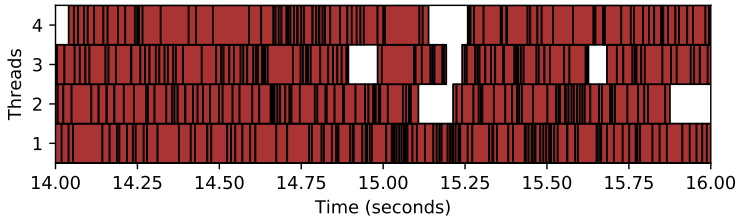
Figure 2.13: LevelDB dependency graph. A directed graph showing replay dependencies enforced by ARTC's resource-aware ordering (solid red) and temporal ordering (dashed blue). Green horizontal edges indicate thread ordering; thus each row of nodes represents a thread. The ordering of the nodes in the horizontal direction is based on their ordering in the original trace. All calls in this window of time are `preads`; each node is labeled with the number of the file descriptor accessed by the call.

ever, what gives ARTC's replay its flexibility is not having slightly *fewer* dependency edges, but much more importantly having far *longer* edges. Measured in time between calls in the original trace, the average temporal-ordering edge is 10ms, whereas ARTC's average edge length is 8.9 seconds.

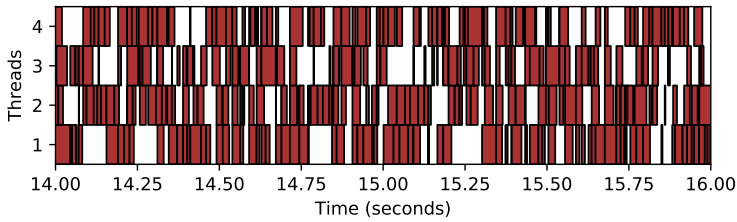
Figure 2.14 shows how enforcing the edges in Figure 2.13 affects when requests are issued during replay. Representative two-second samples are shown for the original program, ARTC replay, and temporally-ordered replay in parts (a), (b), and (c), respectively, run on a single HDD with ext4 with a 4GB



(a) Original program. 3.88 system calls outstanding on average.



(b) ARTC replay. 3.64 system calls outstanding on average.



(c) Temporally-ordered replay. 2.33 system calls outstanding on average.

Figure 2.14: Concurrency. System-call overlap achieved by different replays of a 4-thread LevelDB `readrandom` trace on ext4 with a single HDD.

page cache. For each subfigure, each of the four threads is represented by a row, with grey rectangles indicating spans of time spent in system calls issued by those threads. We observe that in the original program, each thread almost always has an outstanding request, giving the scheduler and disk plenty of flexibility. The replays deviate from this in that some gaps between system calls are visible where the replay threads spent time waiting for ordering dependencies to be satisfied. ARTC, however, shown in Figure 2.14(b), suffers far fewer such stalls than the temporally-ordered replay shown in Figure 2.14(c), achieving 94% of the system-call concurrency shown in Figure 2.14(a), in contrast to temporal ordering's 60%.

2.6 Case Study: Magritte

Here we demonstrate the use of the Magritte benchmark suite to evaluate the relative performance characteristics of two storage systems, using ARTC's detailed output to determine what types of operations dominate thread-time during replay. Thread-time is a measure of time used by individual threads, and will usually be greater than wall-clock time since threads typically run concurrently (for example, two threads running concurrently for two seconds yields four thread-seconds). Fig-

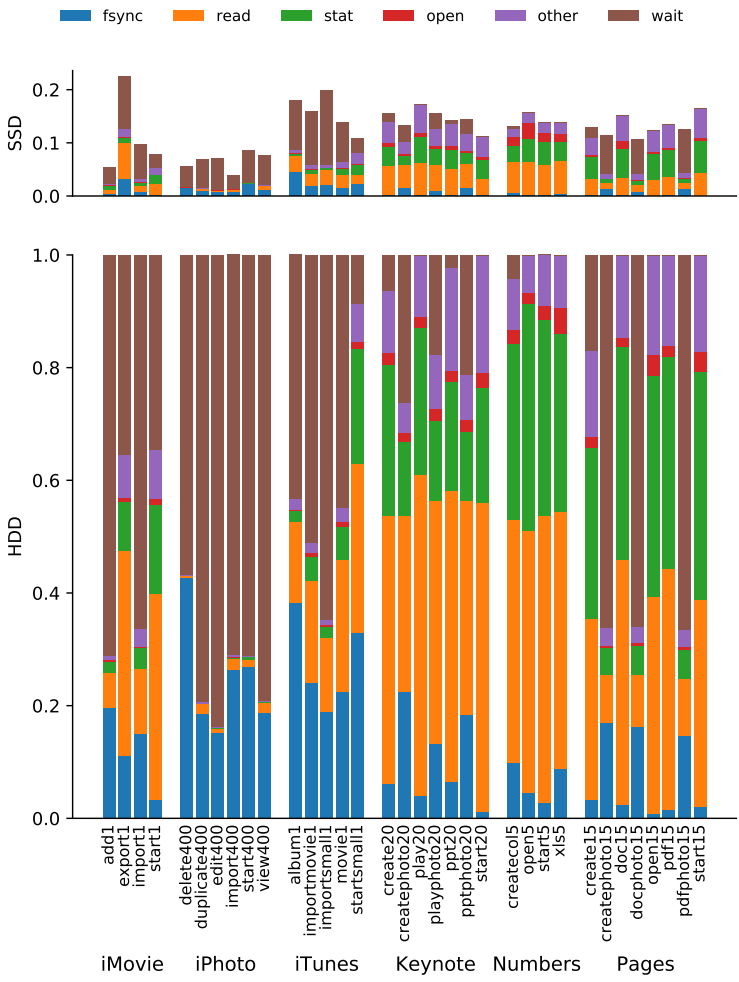


Figure 2.15: Magritte thread-time components on ext4, HDD vs. SSD. The vertical axis of the SSD graph is scaled to match that of the HDD graph.

ure 2.15 shows a breakdown of how thread-time is spent when replaying on a disk and an SSD. Both times are normalized to HDD thread-time.

The SSD plot indicates a thread-time speedup of 5-20× for most applications. Many of the categories with a significant presence for the HDD experiments also have a significant presence on the SSD; however, time spent waiting for `fsync`s is much less significant.

The applications each show distinct patterns. When run on disk, thread time in iPhoto and iTunes tends to be dominated by `fsync`; Numbers and Keynote, on the other hand, are dominated by `read`s and `stat`-family calls (e.g., `stat`, `lstat`, *etc.*). iMovie and Pages are divided across a greater number of categories.

2.6.1 `fsync` Semantics

`fsync` semantics vary across Unix implementations, so on systems where multiple versions are available, ARTC provides an option to select which to use during replay. This capability is particularly useful for cross-platform replay. On Linux, `fsync` typically flushes data to persistent storage, whereas on Mac OS X, `fsync` only flushes data to the storage device, which may merely store the data in a volatile cache. While unusual, OS X's `fsync` implementation does technically conform to the

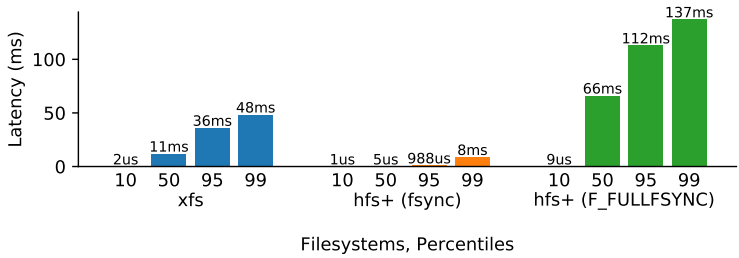


Figure 2.16: `fsync` latency. Latencies are shown at the 10th, 50th, 95th, and 99th percentiles for xfs, and HFS+ with two different `fsync` replay modes.

POSIX definition of `fsync`, which leaves its precise behavior implementation-defined [114]. However, we question the utility of the "become durable at some arbitrary, unknown point in the future" semantics OS X's `fsync` provides, since it does not appear to differ meaningfully from the semantics an application achieves by calling only `write`. To achieve true data safety on OS X, an application must use the non-standard `fcntl(F_FULLFSYNC)` command.

To explore the implications of these two different semantics, we use the `iphoto_delete400` benchmark, which calls `fsync` over 20,000 times. We replay this trace on Linux with xfs, and with both the default and safe (`F_FULLFSYNC`) semantics on Mac OS X with HFS+. Figure 2.16 presents some of the detailed statistics reported by ARTC, which include the

latency of every `fsync` call.

`fsync` on xfs has a median latency of 11ms, about the time necessary for a disk seek. `fsync`s at the 10th percentile return immediately, but this is because iPhoto called `fsync` without performing any writes first, reflecting the tendency of applications to perform unnecessary, inefficient operations [56]. The timings for HFS+ when `fsync` is replayed with default semantics show that latencies are, as expected, clearly too fast for data to be saved persistently; 95% of the calls finish in under 1 millisecond. When replaying the benchmark with the safe semantics, though, latencies are long enough as to not cast doubt on whether the data was durably written.

2.7 Related Work

The use of a compiler to transform multithreaded filesystem traces for replay is somewhat similar to previous work by Joukov *et al.* [72]. Their trace compiler, however, is used primarily as an optimization to reduce runtime processing overhead during replay (which they perform at the VFS level, making it closely tied to operating system specifics). While ARTC's compilation does provide similar benefits, it is more focused on trace analysis and inferring event dependencies. Further, their replay system is designed to preserve the timing

of the original trace, whereas ARTC's entire *raison d'être* is to allow flexibility in that regard.

In other work on I/O trace replay, Anderson *et al.* argue for maximum accuracy, since even slight deviations can produce significant behavioral changes [9]. Tarasov *et al.*, however, argue for merely approximate replay based on general workload characteristics [135]. Our work falls somewhere in between: we replay the exact I/O set in the original trace, though we allow variations in ordering, much like real multithreaded applications. While ARTC may not necessarily produce exactly the same behavior from one run to the next, it more realistically emulates the behavior of real applications (which are likewise not necessarily consistent across runs).

Different approaches have been suggested for mining information from traces. Aguilera *et al.* perform statistical analysis on passive RPC traces to infer inter-call causality [6] for debugging purposes. Mesnier *et al.*'s //Trace uses active tracing, perturbing I/O in order to deduce dependencies between operations [99], and incorporates this information into its replay. ROOT also attempts to infer dependency information from traces, but we rely on hints to glean as much information as possible from a single data point.

Scribe [77] is a replay tool that also partially orders replay events based on resources. Unlike ARTC, Scribe is oriented

more toward debugging and diagnostics than performance analysis, and thus aims for perfect reproduction of the application's in-memory state. This level of detail necessitates intricate platform-specific kernel instrumentation for tracing and replay (which must be done on the same platform), whereas ARTC operates purely with system calls, allowing cross-platform replay and simple trace collection with existing tools.

2.8 Conclusion

Trace replay is a highly useful tool for storage performance analysis. Useful trace replay has been made more difficult, however, by the trends of hardware development leading to increasing CPU core counts and the corresponding increase in the use of multithreading in applications. We have proposed ROOT, a new approach to trace replay that embraces the nondeterminism of multithreaded applications by inferring inter-thread dependency information from a single trace, maximizing the utility of often-scarce trace data. We have presented ARTC, our implementation of ROOT that applies its ideas to Unix system-call traces, and shown that it provides faithful reproduction of a trace's semantics while also achieving accurate performance predictions. With Magritte,

we have also demonstrated how ARTC can be used to automate the generation of realistic benchmark suites. Together, these contributions provide an answer to the question of how to adapt trace replay techniques for the challenges of the multicore era.

3

Storage Virtualization for Solid-State Devices

Storage virtualization has become an important tool in many datacenter and enterprise environments [65, 117, 128]. Via the time-honored technique of adding a layer of indirection, software can flexibly provision storage resources from consolidated hardware, multiplexing it among an array of consumers. This approach provides simpler management and configuration by centralizing it, and improves utilization and overall efficiency by decreasing the waste resulting from over-provisioned hardware [122, 128].

While there have been efforts at flash-oriented updates to virtualization in the storage stack [71], the designs of most existing storage virtualization systems predate flash's

widespread adoption, and are not structured to take full advantage of it. In this chapter we present ANViL, an effort to rethink storage virtualization systems in the context of high performance flash storage hardware. We describe ANViL's design and implementation, with particular attention to the challenges of its internal space management (garbage collection). We also demonstrate how its expanded capabilities can be used to provide not only conventional storage virtualization functionality such as volume snapshots, but also more sophisticated features like file cloning, and atomic commits without the penalty of writing data twice.

3.1 Introduction

Hard disk drives (HDDs) served as the storage workhorse of the computing industry for decades. They provide a simple interface by which software can read and write fixed size blocks of data in a single large, flat array. Their inherent mechanical nature, however – spinning platters and seeking actuator arms – incurs access latencies orders of magnitude longer than than the timescales of CPU operations. A CPU might sit idle for millions of cycles waiting for the drive to position its actuator arm at the right track and its rotate its platters such that the appropriate location on the disk passes

under the read/write head. The slowness of disk access is thus well established as one of the most common bottlenecks constraining overall system performance.

The rise of solid-state storage devices (SSDs) in the last decade, however, has substantially reduced this constraint. While the fundamental storage technology (most commonly NAND flash) has existed for longer, storage capacities were too small and costs too high to make it a viable competitor to the venerable spinning disk. Over time though, flash capacities have grown and costs have decreased [83]; accordingly, SSDs have gradually captured a larger and larger fraction of the storage market, and as of 2018 are commonplace in both consumer computing hardware (as primary storage) and datacenters (as either primary storage or an intermediate layer between DRAM and HDDs [2, 15, 125]).

SSDs offer much lower access latencies than HDDs while filling the same basic role in computing systems, but the fundamental differences in the underlying technology do show through in other ways. In its most raw form, data stored in NAND flash cannot be overwritten directly. Instead, the region of storage must first be explicitly *erased* before being rewritten with new contents. Complicating this process further is the coarse granularity of the erase operation: whereas reads and writes may be performed in units of *pages* (a unit

distinct from a page of virtual memory, but of a comparable size at perhaps 4KiB), the unit of space cleared by an erase operation (an *erase block*) is typically much larger – perhaps 512KiB. Additionally, each such *program/erase* cycle performed incurs physical wear on the storage cells in block it is performed on. Each erase block can thus only endure a limited number of program/erase cycles before it fails permanently and must be taken out of service.

If exposed directly to system software, these additional complications would render flash storage incompatible with existing software written for the simpler HDD interface, requiring a large amount of code to be rewritten and thus presenting a major barrier to the adoption of the newer, faster technology. To sidestep this problem, most SSDs incorporate a *flash translation layer* (FTL) – a piece of on-device firmware that keeps the flash-specific complexity internal to the SSD and presents a simpler HDD-style read/write interface to the host system.

By providing this convenient abstraction, FTLs have allowed SSDs to be easily integrated into existing storage stacks while requiring little to no modification of software. However, while FTLs provide the necessary compatibility shim, the different characteristics of SSDs can nevertheless leak through, often manifesting as undesirable performance variations in

applications that do not exhibit "flash-friendly" access patterns [59]. Thus, while *compatibility* can be easily achieved, fully exploiting the potential of newer storage technologies still requires restructuring of some software in the storage stack to better match the properties of the underlying hardware.

As the trend of flash storage increasing in capacity and decreasing in cost continues, ever-greater quantities of data are being stored in flash, which in turn drives increasing demand for storage features and functionality like those found in traditional disk-based storage systems. Prior work has observed the impact of flash on storage architectures while also noting that flash presents new challenges in the implementation of classic storage system features and the expectations placed on them [75, 79, 125, 131, 161].

At the same time, studies have observed that flash presents an opportunity to rethink the overall architecture of the I/O stack, with designs that reuse powerful primitive functions to create composable data services [1, 71, 88, 90, 108, 125]. For example, studies such as FlashTier [125], NVMKV [90] and DFS [71] demonstrate that log-structured stores, which are already well-suited to flash, can also provide address-mapping capabilities which facilitate the implementation of applications and common data services (such as snapshots) with relatively little effort and minimal redundancy in the I/O stack.

Address-mapping in storage systems fits well as a major component of storage virtualization, a piece of the storage stack ripe for modernization for the flash era. Virtualization of many forms has been widely employed as a technique for managing and exploiting the available resources in computing systems, from memory and processors to entire machines [3, 10, 12, 25, 38, 49, 116]. Virtual memory in particular has enabled numerous features and optimizations, including the `mmap(2)` interface to file I/O, shared libraries, efficient `fork(2)`, zero-copy I/O, and page sharing between virtual machines [11, 145].

Storage virtualization, however, while conceptually similar to memory virtualization, has typically been of limited use to applications, focusing instead on storage management by introducing an abstraction between the physical storage layout and the logical device as presented to a host or application using it [40, 58, 141]. Features and functionality enabled by storage virtualization, such as deduplication, replication, and thin-provisioning, remain hidden behind the block device interface. While highly useful, the features of existing storage virtualization systems are primarily limited to administrative functionality, such as defining and provisioning volumes, offering nothing to actual applications beyond standard read and write operations. As others have shown, these limitations in storage virtualization result in sub-optimal application perfor-

mance and duplication of functionality across different layers in the storage stack [37, 46, 91, 108].

Some of the limits of storage virtualization have been addressed in recent research on FTLs, with new machinery proposed to support features such atomic writes, persistent trim, and sparse addressing [80, 91, 104, 108, 125]. These extensions enable applications to better leverage the virtualization already built into the FTL and also enable the removal of redundant functionality across system layers, resulting in improved flash write endurance and application-level performance [71, 108].

We propose a simple yet powerful set of primitives based on *fine-grained address remapping* at both the block and extent level. As we will show, fine-grained address remapping provides the flexibility needed to benefit applications while still retaining the generality necessary to provide the functionality offered by existing virtualized volume managers. By allowing the host to manipulate the block-level logical-to-physical address map with *clone*, *move*, and *delete* operations, we enable storage virtualization to more closely resemble virtualized memory in its fine-grained flexibility and broad utility, though in a manner adapted to the needs of persistent storage.

We illustrate the utility of our approach by developing the Advanced Non-volatile storage Virtualization Layer (ANViL),

a prototype implementation of fine-grained address remapping as a stacking block device driver, to efficiently implement both file and volume snapshots, deduplication, and single-write journaling. More specifically, we demonstrate how ANViL can provide high performance volume snapshots, offering as much as a 7× performance improvement over an existing copy-on-write implementation of this feature. We show how ANViL can be used to allow common, conventional filesystems to easily add support for file-level snapshots without requiring any radical redesign. We also demonstrate how it can be leveraged to provide a performance boost of up to 50% for transactional commits in a journaling filesystem.

We also address in detail one of the foremost challenges of implementing ANViL, namely that of space management (garbage collection). The combination of large scale, high performance requirements, and the feature set provided by ANViL make the task of tracking exactly what data is and is not referenced (and reclaiming space from data that is not) a difficult one. ANViL's garbage collection (GC) employs a novel approach to tackle this problem, borrowing ideas from the world of programming language implementations and adapting them to the domain of storage systems.

3.2 Background

Existing storage virtualization systems focus their feature sets primarily on functionality "behind" the block interface, offering features like replication, thin-provisioning, and volume snapshots geared toward simplified and improved storage administration [40, 141]. They offer little, however, in the way of added functionality to the *consumers* of the block interface: the filesystems, databases, and other applications that actually access data from the virtualized storage. Existing storage technologies, particularly those found in flash devices, offer much of the infrastructure necessary to provide more advanced storage virtualization that could provide a richer interface directly beneficial to applications.

At its innermost physical level, flash storage does not offer the simple read/write interface of conventional hard disk drives (HDDs), around which existing storage software has been designed. While reads can be performed simply, a write (or *program*) operation must be preceded by a relatively slow and energy-intensive *erase* operation on a larger erase block (often hundreds of kilobytes), before which any live data in the erase block must be copied elsewhere. FTLs simplify integration of this more complex interface into existing systems by adapting the native flash interface to the simpler HDD-style read/write interface, hiding the complexity of program/erase

cycles from other system components and making the flash device appear essentially as a faster HDD. In order to achieve this, FTLs typically employ log-style writing, in which data is never overwritten in-place, but instead appended to the head of a log [121]. The FTL then maintains an internal address-remapping table to track which locations in the physical log correspond to which addresses in the logical block address space provided to other layers of the storage stack [53, 132].

Such an address map provides the core machinery that would be necessary to provide more sophisticated storage virtualization, but its existence is not exposed to the host system, preventing its capabilities from being fully exploited. A variety of primitives have been proposed to better expose the internal power of flash translation layers and similar log and remapping style systems, including atomic writes, sparse addressing (thin provisioning), persistent TRIM, and cache-friendly garbage collection models [91, 104, 108, 125, 156]. These have been shown to be valuable to a range of applications from filesystems to databases, key-value stores, and caches.

3.3 Structure

ANViL is a layer incorporated into the block level of the storage stack. Much like software RAID or the Linux device-mapper subsystem [57], it presents a virtual block device for use by layers above it in the storage stack, and itself runs on top of another lower-level block device (such as a bare SSD or a RAID array of SSDs).

The block device it presents exposes a 48-bit logical block address space, yielding 128PiB with a 512-byte block size. ANViL maps portions of this address space to corresponding regions of the physical block address space provided by the backing device beneath it. This mapping is done at block granularity, combining contiguous regions into a single extent for data in multi-block write requests.

A given logical address can be either mapped or unmapped. A read of a mapped address returns the data stored at the corresponding physical address. A read of an unmapped address simply returns a block of zeros, much like a read of a hole in a sparse file. Write requests are handled in a redirect-on-write fashion, detailed later in Section 3.5.1.

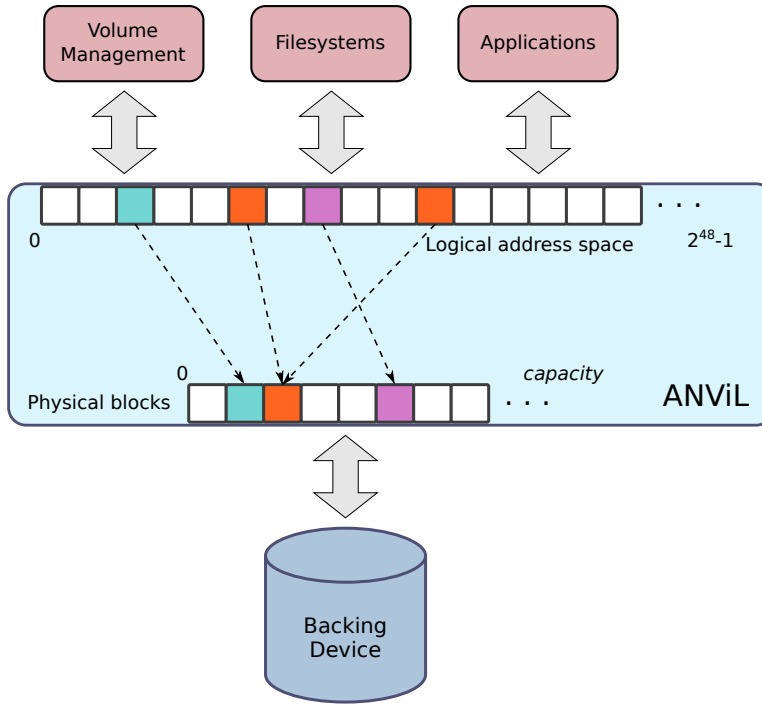


Figure 3.1: ANViL's position in the storage stack. While the backing device used to provide ANViL's physical storage space is not required to be flash, it is explicitly designed to operate in a flash-friendly manner and is intended for use with SSDs (or arrays thereof).

3.4 Interfaces

Address-remapping structures exist in FTLs and storage engines that provide thin provisioning and other storage virtualization functions today [4, 53]. While ANViL's flash-oriented, log-structured design is the underlying reason for the existence of this remapping machinery, going a step further and exposing it to applications and filesystems is a key design decision that allows it to significantly expand the functionality provided to higher-level software by its storage stack, enabling straightforward implementation of features like file cloning and efficient atomic transactions.

In this section we describe the range operations via which ANViL allows direct manipulation of its internal address map, and a set of complementary properties that enhance their general utility and applicability.

3.4.1 Range Operations

ANViL's interface augments the traditional block-I/O read and write operations with three additional range operations: clone, move, and delete.

Range clone: `clone(src, len, dst)`: The range clone operation instantiates new mappings in a given range of logical address space (the *destination* range) that point to the

same physical addresses mapped at the corresponding logical addresses in another range (the *source* range); upon completion the two ranges share storage space. A read of an address in one range will return the same data as would be returned by a read of the corresponding address in the other range. This operation can be used to quickly relocate data from one location to another without incurring the time, space, and I/O bandwidth costs of a simplistic read-and-rewrite copy operation. A range clone applied to the logical address space providing storage for a volume can thus be used to easily implement a volume-snapshot feature. Similarly, a filesystem need only internally allocate a corresponding region of logical address space and issue a range clone of a file's data blocks to provide a space- and I/O-efficient file-snapshot operation. (These use-cases are examined in greater detail in Section 3.7.)

Range move: `move(src, len, dst)`: The range move operation is similar to a range clone, but leaves the source logical address range unmapped. This operation has the effect of efficiently transferring data from one location to another, again avoiding the overheads of reading in data and writing it back out to a new location. In combination with the durability and atomicity properties described in Section 3.4.2, this provides a mechanism via which transactional storage systems such as relational databases and journaling filesystems can implement

an efficient transaction commit protocol that does not require writing transaction data twice (see Section 3.7).

Range delete: `delete(src, len)`: The range delete operation simply unmaps a range of the logical address space, effectively deleting whatever data had been present there. This operation is similar to the `TRIM` or `DISCARD` operation offered by existing SSDs. However, unlike `TRIM` or `DISCARD`, which are merely advisory, the stricter range delete operation guarantees that upon acknowledgment of completion the specified logical address range is persistently unmapped. Range deletion is conceptually similar to the persistent TRIM operation defined in prior work [70, 104]. In ANViL it is additionally intended to be used in tandem with the range clone operation for features such as snapshot management (so that existing snapshots can be removed when no longer needed).

3.4.2 Complementary Properties

While giving the host system the ability to manipulate the storage address map is the primary aim of our proposed interface, other properties complement our interfaces nicely and make them more useful in practice for real-world storage systems.

Sparse addressing (thin provisioning): In conventional storage devices, the logical space exposed to the host system is mapped one-to-one to the (advertised) physical capacity of the device. However, the existence of the range clone operation implies that the address map must be many-to-one. Thus, in order to retain the ability to utilize the available storage capacity, the logical address space must be expanded beyond the actual storage capacity of the device – in other words, the device must be *thin-provisioned* or *sparse*. The size of the logical address space, now decoupled from the physical capacity of the device, determines the upper limit on the total number of cloned mappings that may exist for a given block.

Durability: The effects of a range operation must be crash-safe in the same manner that an ordinary data write is: once acknowledged as complete, the alteration to the address map must persist across a crash or power loss. This requirement implies that the metadata modification must be synchronously persisted, and thus that each range operation implies a write to the underlying physical storage media.

Atomicity: Because it provides significant added utility for applications in implementing semantics such as transactional updates, we propose that a vector of range operations may be submitted as a single atomic batch, guaranteeing that

after a crash or power loss, the effects of either *all* or *none* of the requested operations will remain persistent upon recovery. Log-structuring (described in Section 3.5.1) makes this relatively simple to implement.

3.5 Implementation

In this section we describe the implementation of our prototype, the Advanced Non-volatile storage Virtualization Layer (ANViL), a Linux kernel module that acts as a generic stacking block device driver. ANViL runs on top of single storage devices as well as RAID arrays of multiple devices and is equally at home on either. It is not a full FTL, but it bears a strong resemblance to one. Though an implementation within the context of an existing host-based FTL would have been a possibility, we chose instead to build ANViL as a separate layer to simplify development.

3.5.1 Log Structuring

In order to support the previously-described set of operations (Section 3.4), ANViL is implemented as a log-structured block device. Every range operation is represented by a note written to the log specifying the point in the logical ordering of updates at which it was performed. The note also records the

alterations to the logical address map that were performed; this simplifies reconstruction of the device's metadata after a crash.

Each incoming write is redirected to a new physical location, regardless of whether the written-to logical address had been mapped or unmapped. Updates to a given logical range thus do not affect other logical ranges which might share physical data; the written address is decoupled from the physical block containing the shared data while the other logical addresses mapped to it retain that mapping.

Similarly to LFS [121], physical space on the backing device is managed in large segments (ANViL's default to 128MiB). Each individual segment is written sequentially and a log is maintained that links them together in chronological order. Once a segment has been fully written, it is made immutable.

3.5.2 Metadata Persistence

Whenever ANViL receives a write request, before acknowledging completion it must store in non-volatile media not only the data requested to be written, but also any updates to its own internal metadata necessary to guarantee that it will be able to read the block back even after a crash or power loss. The additional metadata is small (24 bytes per write request,

independent of size), but due to being a stacked layer of the block I/O path, writing an additional 24 bytes would require it to write out another entire block. Done naïvely, the extra blocks would incur an immediate 100% write amplification for a workload consisting of single-block writes, harming both performance and flash device lifespan. However, for a workload with multiple outstanding write requests (a write I/O queue depth greater than one), metadata updates for multiple requests can be batched together into a single block write, amortizing the metadata update cost across multiple writes.

ANViL thus uses an adaptive write batching algorithm, which, upon receiving a write request, waits for a small period of time to see if further write requests arrive, increasing the effectiveness of this metadata batching optimization, while balancing the time spent waiting for another write with impact on the latency of the current write.

3.5.3 Space Management

Space on the backing device is allocated at block granularity for incoming write requests. When a write overwrites a logical address that was already written and thus mapped to an existing backing-device address, the new write is allocated a new physical address on the backing device and the old mapping for the logical address is deleted and replaced by a

mapping to the new backing device address. When no mappings to a given block of the backing device remain, that block becomes "dead" or invalid, and its space may be reclaimed. However, in order to maintain large, contiguous regions of free space in the backing device so as to allow for sequential writing, freeing individual blocks as they become invalid is not a good approach for ANViL. Instead, the minimum unit of space reclamation is one segment (which functions somewhat analogously to an erase block in an FTL).

A background garbage collector continuously searches for segments of backing device space that are under-utilized (i.e. have a large number of invalid blocks). When such a segment is found, its remaining live blocks are copied into a new segment (appended at the current head of the log as with a normal write), any logical addresses mapped to them are updated to point to the new location they have been written out to, and finally the entire segment is returned to the space allocator for reuse. Achieving effective garbage collection is critically important and was one of the primary challenges in implementing ANViL; its design is discussed in detail later in the following section.

3.6 Garbage Collection

This section details the design and implementation of ANViL's garbage collector (GC). The requirements for ANViL's GC are different than those of a conventional SSD, primarily due to its many-to-one address map [121]. A GC for a traditional log-structured storage system like the one described in LFS is simple, with each block referenced by at most one logical location. Since ANViL aims to support much richer functionality, a single physical data block may be referred to by more than one logical address, with the number of references to a single physical location ideally limited only by the available physical storage capacity. We now outline the major factors in the design of the ANViL garbage collector and discuss why traditional GC techniques are not directly applicable.

3.6.1 Design Considerations

Capacity scaling: The capacities of modern storage systems are continually growing, and now often offer many terabyte (if not a petabyte or more) of storage. The ability to scale gracefully to large storage capacities is thus a requirement for the ANViL GC.

Reference scaling: Heavy use of the advanced storage virtualization capabilities offered by ANViL's range operations can result in large numbers of references to physical data blocks. For example, a user of a storage array might wish to retain nightly snapshots of a volume for backup or auditing purposes, resulting in many repeated references to the same underlying physical data blocks for infrequently-modified files. We do not wish to artificially limit the extent to which these features can be used, so it is important that ANViL's GC be able to handle data with essentially arbitrarily many references. It must also not impede the instantiation of new references to existing data as it operates.

Performance predictability: Performance is improving with every generation of non-volatile memory devices, with a single modern flash drive capable of delivering hundreds of thousands to millions of I/O operations per second (IOPS). Moreover, users and applications expect *predictable* performance from storage systems; the ANViL GC should thus strive to avoid incurring unpredictable fluctuations in performance. Additionally, background GC activity must be able to keep up with the rate of foreground operations so as not to accumulate a backlog of pending space-reclamation work.

Memory consumption: Memory is always a precious resource and the design of the GC must to be conscientious in its use of it. The design should be able to handle large-scale storage systems (in both capacity and reference count) without requiring enormous quantities of RAM. Frugality with memory is especially necessary if the design is to be applicable in an "off-load" device in which an ANViL-like layer were implemented in device firmware instead of in the host system's OS. The GC's design thus may need to make compromises that trade off CPU and GC efficiency against memory consumption where necessary.

3.6.2 Possible Approaches

There are many different ways of implementing garbage collection for log-structured storage systems. We now examine some existing approaches and explain their applicability (or lack thereof) in the context of ANViL.

Bitmaps

Bitmaps, a time-honored strategy for space-management in storage systems [96, 121], are perhaps the most obvious potential approach to GC. With bitmaps, tracking which blocks are in use and which are free is straightforward. While bitmaps are efficient in both memory consumption and CPU utilization,

they are insufficient to track the in-use/free status of physical blocks in the context of ANViL's many-to-one address map. For example, a simple set-on-map, clear-on-unmap bitmap-management algorithm would be inaccurate if one were to simply clone a live block's mapping to a new logical address and then unmap the original address (the block would have a live reference but its bitmap state would incorrectly indicate it as being free).

Reference Counting

Alternatively, a garbage collector could employ an array of reference counts to track the number of mappings to each block. In fact, a bitmap is simply a special case of a reference count array with single-bit (saturating) reference counts. If we generalize the bitmap approach to use multi-bit reference counts, we can address the inaccuracy problem inherent to bitmaps tracking a many-to-one address map, using a simple increment-on-map, decrement-on-unmap reference count management algorithm. This approach, however, raises a follow-on question to which there is no clear, obviously-correct answer: how large should these reference counts be? Larger reference counts require more memory to store, but smaller ones impose undesirable limitations on the use of the special features offered by ANViL. Further, even setting aside this particular

question, reference counts still do not address a significant need for the ANViL GC. The GC in a multi-reference log-structured system must be able to determine not only how many references to a given physical block exist, but also *where* those mappings are in the logical address space so that it can update them after copying data to a new location. Regardless of their size, reference counts simply cannot provide this information, meaning that in addition to its expense in DRAM consumption, this would be at best an incomplete solution.

Reverse Map

To overcome the limitations of reference counts, one could expand the GC's metadata-tracking to use a full reverse map (mapping each physical address in the backing device to the set of all logical addresses that are mapped to it) in addition to the primary forward map structure. This strategy would provide all the information provided by reference counts and, depending on its exact implementation, would likely avoid imposing arbitrary limits on the number of references to a given block. Most importantly, a full reverse map would also be able to supply the necessary information for the GC to update the (forward) address map after moving data to a new physical location. However, a reverse map would require at least as much additional DRAM space as the forward

map, and likely more, since the data structure mapped to by each physical address would be a set that would have to support reasonably efficient insertion and deletion. The cost of implementing this would simply be unacceptably high in terms of DRAM consumption in addition to the extra bookkeeping work it incur in the performance critical foreground I/O path to keep the reverse map up to date.

Mark and Sweep

Mark and sweep is a garbage collection approach in the category of *tracing* GCs [147]. Tracing collectors determine the liveness of data by evaluating its reachability starting from a set of *roots*. Tracing GC is most widely known for its application in the context of programming language implementations, such as Java virtual machines and interpreters for dynamic languages [48, 140]. In these collectors, the managed data items are allocated objects in memory and the reachability graph is determined by following pointers starting from a set of root pointers on the stack and in global memory.

Mark and sweep, as its name suggests, consist of two phases. In the *mark* phase, the collector performs a complete reachability analysis on the entire object graph. In programming-language GCs, this involves following all pointers in the root set and recursively continuing with pointers

within the pointed-to objects, marking each object traversed in this manner. This marking determines the entire set of transitively reachable (and thus live) objects; any object not in this set is thus "dead" (unreferenced). Once the mark phase is complete, the ensuing *sweep* phase then simply reclaims all unmarked objects.

In addition to programming-language GCs, however, mark and sweep has also been explored in the context of storage systems [28, 52, 73]. Deduplication systems, for example, have used mark and sweep to improve single node scalability [52], and BigTable employs a mark and sweep based garbage collector to cleanup its SSTables [28]. Likewise, despite being a storage system, ANViL's feature set gives it some properties (most notably the potential for data items with large numbers of references) that resemble those of programming-language runtimes.

ANViL's garbage collector thus takes a hybrid approach that is based on the mark and sweep strategy, but augments it with bitmaps to aid in selecting reclamation targets, and partial, ephemeral reverse maps to provide it with the information necessary to relocate data while avoiding the excessive memory consumption of a full reverse map.

3.6.3 Design

ANViL's GC is, at its core, a mark and sweep based collector, though the reference graph that it traverses has a simpler structure than the reference graphs found in language runtimes. The root set consists of all the mapped addresses in the logical block address space, but because physical data blocks cannot contain pointers directly to other physical data blocks, no recursion is needed in the traversal of the graph. A physical block may contain references to other data blocks (as would be found in filesystem metadata, for example, where an inode contains pointers to a file's data blocks), but such references can only exist via *logical* addresses, because the physical address space is entirely internal to ANViL and is not visible to higher levels of the storage stack. Because all logical addresses are already in the root set, physical blocks transitively referenced by them would already be found by the mark phase anyway, so a single step from a logical address to the corresponding physical address is all that is needed (a scan of the block's data to search for additional pointers is not necessary).

The ANViL GC is thus split into two primary components, which we call the *scanner* and the *cleaner*, mirroring the mark and sweep phases, respectively. We use different terminology for these components because in ANViL they are not separate

phases executed in series, but actually both run continuously and concurrently; their operation is detailed in Sections 3.6.4 and 3.6.5.

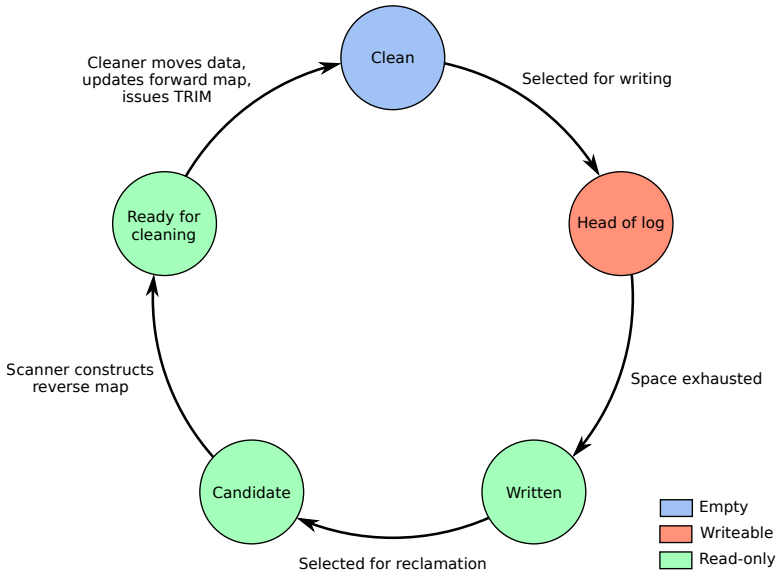


Figure 3.2: Segment life cycle. Segments in the states shaded green are immutable and managed entirely by the GC; *written* and *candidate* segments are managed by the scanner while those in the *ready for cleaning* state are managed by the cleaner.

Figure 3.2 provides a high-level illustration of the cycle of segment-granularity space management in the ANViL GC.

Segments start out in the *clean* state, available for use and containing no valid data. When one is selected to receive incoming data from a write request, it becomes the head of the log, and is written sequentially until full. When completely filled with data, it is handed off to the GC and another segment is selected as the new head of the log. Depending on the amount of data in the segment that becomes invalid over time, it may eventually become a potential reclamation target (a *candidate*). If it is selected as a candidate, the next scan cycle will construct a reverse map for it, after which it is ready for cleaning. It is then handed off to the cleaner, which copies its remaining live data forward into a new segment (whichever is the head of the log at that point in time), updates the forward map to refer to the new locations of the moved blocks, and finally performs a TRIM operation on the entire segment. At this point the segment is clean again and is returned to the pool of free space. At any given time, most segments in the system will be in either the *clean* or *written* states, and exactly one segment will be the current head of the log.

3.6.4 Scanner

The task of the scanner is to select and prepare segments (the contiguous 128MiB regions in which ANViL manages physical space) for reclamation. Segments to be garbage collected may

contain both valid and invalid data. Ideally, segments selected to be reclaimed would be empty or nearly so, as this minimizes the amount of data that must copied forward, reducing write amplification [121]. ANViL's scanner consists of a set of background threads that periodically traverse the forward address map, inspecting mappings of valid data blocks to select segments for potential cleaning. This work is split into two phases, *candidate selection* and *candidate preparation*.

The first phase of the scanner, candidate selection, scans through the forward map to identify segments that fall below the desired data-validity threshold (the number of data blocks within them that are referenced and thus still live). This task is accomplished using a bitmap for each segment. These bitmaps start out with all bits clear at the beginning of the candidate selection scan cycle. For each valid mapping encountered during the traversal of the forward map, the scanner sets a bit in the corresponding segment's bitmap indicating that the block referenced by that mapping is in use. The bitmaps constructed during this phase are shown in Stage 2 of Figure 3.3. At the end of the pass, the number of set bits in each segment's bitmap gives an indication of how much valid data remains in that segment.

This metric is not necessarily completely accurate, because concurrent foreground operations (such as overwrites or range

deletes) that occur during the scan cycle can cause blocks that were live at the beginning of the scan to become invalid by the end of it. It does provide an upper bound on data validity, however, because blocks that are invalid cannot become live again until after the containing segment has been fully garbage collected and released back to the free space pool by the GC. A mapping to a physical block (of which one or more must exist for the block to be live) can only be instantiated by a write or a clone or move range operation. Writes are always directed to the segment at the head of the log, which is not tracked by the GC. Range operations operate purely within the logical-address namespace and thus can only refer to physical blocks indirectly via logical addresses mapped to them. An invalid block (one with no mappings in the logical address space) in a GC-tracked segment thus cannot be affected by any foreground operations.

The second phase of the scanner, candidate preparation, constructs reverse maps for each selected candidate segment. It performs another full pass of the forward address map; when it encounters a mapping whose physical data block resides in a segment that has been selected as a candidate for cleaning, it inserts the logical address into the segment's reverse map, adding it to the set of addresses mapped to that physical block. This phase is shown in Stage 3 of Figure 3.3.

The scanner ultimately determines the overall write amplification introduced in the system: a poor choice of candidate segments may lead to inefficient space reclamation as well as device wear-out caused by excessive writing. The scanner also implicitly limits the throughput of the cleaner (which performs the actual reclamation of space): if the scanner is not producing segments selected and prepared for collection, the cleaner cannot reclaim any space. The speed and the accuracy of the scanner are thus critical to ANViL's operation.

3.6.5 Cleaner

The second component of ANViL's GC, the cleaner, is responsible for the actual reclamation of unused space in the segments selected by the scanner. For each segment the scanner prepares for collection, the cleaner must move all valid data remaining in the segment to a new physical location and then update the forward map accordingly. The cleaner divides this process into three steps.

Copy-forward: The cleaner must relocate all valid data (if any) in a candidate segment to a new location on the log before it can reclaim the segment. To relocate data, the cleaner first issues reads to all valid data blocks identified by the scanner. These reads can be issued in parallel, taking advantage of the

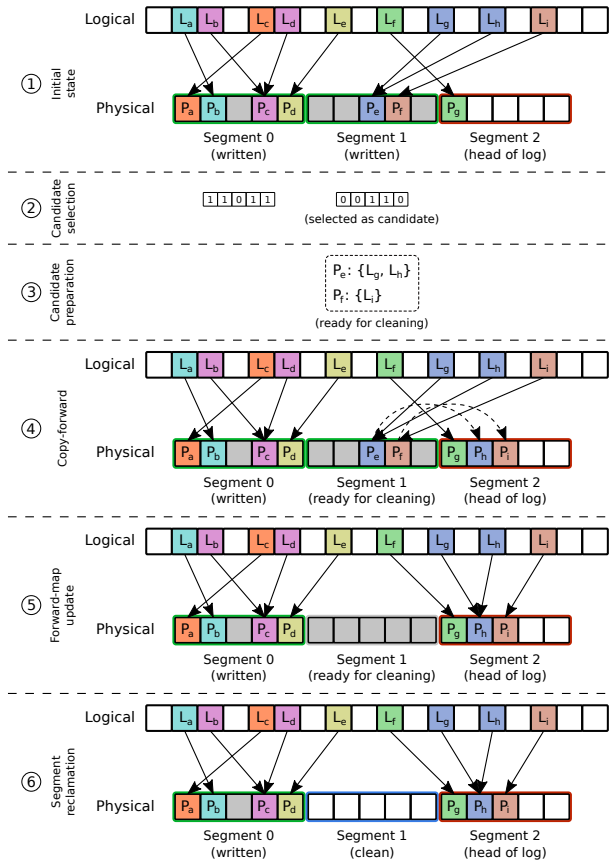


Figure 3.3: The ANViL garbage collection process. Starting from the initial state in ①, ② through ⑥ illustrate the actions of the scanner and the cleaner in reclaiming a segment.

high degree of internal parallelism offered by high-performance flash storage devices. When these reads complete, the cleaner allocates new space at the current head of the log and writes the data out at this new locations. During this step the data blocks being moved remain valid and available in their original locations, so concurrent foreground reads can still safely access them there. This process is shown in Stage 4 of Figure 3.3.

Forward-map update: The cleaner updates the forward address map only after the segment's valid data has been successfully written out to its new location. In the window of time between the completion of the write to the new location and the segment being freed (which only occurs after all forward-map updates have been performed), both the old and new locations of the data are valid and either may be safely used to service reads. Thus, even with multiple discontiguous blocks of valid data to be moved, the forward map can be updated one mapping at a time without introducing any gaps during which invalid data could be seen, or having to lock out foreground I/O requests. The results of this step are shown in Stage 5 of Figure 3.3.

Segment reclamation: After the forward map has been fully updated for all the valid data blocks within the segment

being cleaned, the cleaner issues a TRIM request to the backing device for the segment's physical space and finally returns it to the space allocator for reuse. While the TRIM is not strictly necessary, it helps to lighten the workload of the internal garbage collection in the FTLs of the underlying flash devices providing ANViL's backing storage. After the TRIM operation is performed, the segment's old data no longer exists and any reference to a data block within the segment would be invalid; this possibility is avoided by delaying the TRIM operation until all forward-map updates (and any outstanding foreground read requests that may have been issued to the region) have completed. This step produces the state depicted in Stage 6 of Figure 3.3.

3.6.6 Techniques and Optimizations

While the description above outlines the general structure of the ANViL GC, its implementation incorporates a number of additional features; these are described in the following subsections.

Multithreaded Scanning

The work of a scan cycle is entirely CPU-bound (it performs no I/O) and potentially large, due to ANViL's vast logical address space into which physical storage can be mapped. It

is amenable to parallelization though, and hence the scanner is multithreaded, taking advantage of the large numbers of processor cores available in recent generations of CPUs. Each thread is given a subset of the logical address space to scan. Exactly how to partition the logical address space among these threads, however, is a somewhat more difficult question than it might at first appear. The logical address space is sparsely populated, and the scanner only traverses addresses that are actually present in the forward map. In order to spread work evenly among scanner threads, each thread should scan approximately the same number of mappings. The scanner, however, has no high-level overview of the distribution of mapped addresses within the logical address space and as such it is not trivial to divide up the work into equal-sized parts when beginning a scan cycle.

To address this issue, the scanner employs a dynamic work reassignment algorithm. The key insight enabling this algorithm is that there is no actual need for the division of logical address space between threads to be statically determined at the start of each scan cycle. When any thread finishes its assigned work, it sets a global flag requesting that the remaining scanning work be redistributed. Each running thread checks this flag periodically, and upon observing it being set, records the progress it has made in its own assigned portion of the

address space and then waits at a barrier. When all threads have reached the barrier, a designated leader thread then re-partitions the remaining work to distribute parts of it to any idle threads. The scanner threads are then released from the barrier and begin scanning their newly-reassigned portions of the address space, repeating the reassignment process when any threads finish their work, until all populated regions of the logical address space have been scanned.

Pipelined Scanning

While the scanner is split into two phases and the second phase (candidate preparation) is dependent on the first (candidate selection), this dependency only exists for each individual segment. Thus, as a performance optimization, the two scan phases are pipelined – that is, they are run concurrently for different segments. On any given scan cycle (full traversal of the forward address map), the scanner can be performing the work of the first scan on one set of segments and the work for the second on another (disjoint) set of segments, effectively pipelining them.

Pipelined scanning does increase the "latency" of the reclamation of any individual segment, since it must take two complete trips through the scanner, and each of these trips is slightly slower due to combining the work of the two phases.

Latency is not an important metric for ANViL, however. GC throughput is much more critical, and is aided by pipelined scanning, because the cleaner can be provided with newly prepared segments for reclamation at the end of every scan cycle instead of only every other cycle.

Selective Segment Tracking

The scanner is responsible for constructing reverse maps for each candidate segment, which are used by the cleaner in the process of reclaiming the selected segments. As discussed earlier in this section, reverse maps are expensive; a full-system reverse map would incur significant memory bloat. It is thus important to control the memory consumption of these maps, which is affected by not only *how many* segments are selected as candidates, but also *which* specific segments are chosen.

The amount of memory required for a given segment's reverse map is a function of how many data blocks in the segment are valid and how many logical mappings exist that refer to those blocks. The bitmaps built by the scanner during its candidate-selection phase provide an upper bound on the number of valid data blocks (though they do not provide any indication of how many mappings to them it encountered). In order to control memory consumption, the scanner thus

limits the total number of segments it selects as candidates, preferring those with the least amount of valid data within them. In addition to reducing the memory consumed by candidate segment reverse maps, this also reduces the amount of I/O that must be done by the cleaner in its copy-forward step to relocate the valid data out of the segment before freeing it.

GC Notifications from Foreground I/O

Because the ANViL garbage collector operates concurrently with normal I/O activity, it is entirely possible that foreground operations can invalidate information recorded by the GC as it prepares to reclaim a segment. ANViL thus inserts "hooks" into the foreground I/O paths for writes and range operations to notify the GC of any changes made – this is directly analogous to the write barriers used in some programming-language GCs [147]. We use these hooks both for maintaining correctness and for a small optimization to reduce write amplification.

Once the scanner has selected a segment as a candidate and then completed the subsequent pass to construct the reverse map for it, the segment waits for some period of time to be processed by the cleaner (which may still be busy reclaiming other segments from a previous pass). If in that window

of time any changes are made to the set of logical addresses mapped to physical blocks in that segment, however, the segment's reverse map becomes stale. If new mappings to existing data are added via a range clone, those mappings (being absent from the reverse map) would not be properly updated by the cleaner, and thus would refer to invalid locations after the segment is freed. Similarly, existing mappings removed by a range delete operation would be incorrectly reinstantiated by the cleaner if the corresponding entries in the segment's reverse map were still present. For this reason, the code paths of foreground operations that mutate the forward map include hooks (our form of write barriers) to perform the necessary corresponding update to the GC's data structures. The hook functions check if any affected physical block addresses belong to a segment that has been selected for cleaning, and if so perform the necessary updates to that segment's reverse map.

While maintaining semantic correctness is the most critical function of these notifications, they also provide an opportunity for a small optimization in the GC that can help to eliminate unnecessary writes to the backing device. Garbage collection unavoidably leads to some degree of write amplification in log-structured storage systems; a large body of existing work describes various techniques to reduce it [121, 154, 155]. With the structure of ANViL's GC, however, the same vulner-

able window of time described above can also lead to needless I/O by the cleaner. If a write or range delete were to remove the last remaining mapping to a previously-valid physical data block, any I/O done by the cleaner to read its contents and re-write them to a new location would be wasted, since there would be no mappings to it remaining in the forward map (it is no longer live). By avoiding such unnecessary I/O, the updates to the GC's reverse maps via the hooks in the foreground operation paths can also reduce write amplification, improving both performance and increasing the lifespan of the underlying flash device.

Concurrency and Rate Limiting

A major design goal of the ANViL GC is to be as concurrent as possible, strongly preferring some amount of continuous background activity to outright pauses for garbage collection, avoiding the "stop-the-world" approach sometimes employed in programming-language GCs. The cleaner's forward-copying of valid data in segments selected for cleaning necessarily interferes with foreground I/O traffic by consuming some of the backing device's available bandwidth. ANViL limits the impact of this activity using explicit rate-limiting of the GC's I/O. The job of the rate limiter is to decide what fraction of the total available bandwidth should be granted to GC

activity, and to then enforce that limit.

The fraction of I/O bandwidth the rate limiter allows the GC to use is a function of the total space utilization in the system, measured as a segment-granularity fraction of the capacity of the backing device (the number of segments not currently free divided the total number of segments). There are two key thresholds in this metric. The first is a simple activation threshold below which all GC activity is disabled (our experiments have put this threshold at 50%). Once overall space utilization rises above this level, garbage collection is enabled and granted a fraction of the backing device's I/O bandwidth that increases with increasing space utilization. The intent of this design is that as utilization increases the system will reach a stable equilibrium point at which the rate of the GC's space reclamation is well-matched to the rate of incoming write traffic. And while the GC and rate-limiter are designed to avoid this situation, there does exist a second threshold, when ANViL's available physical storage capacity is all but completely exhausted, at which point it will as a last resort enter a "panic" mode that actually halts foreground write traffic so as to allow the GC to use all available I/O bandwidth while it attempts to recover and return to normal operation.

3.7 Case Studies

Here we demonstrate the generality and utility of ANViL and its range operations by implementing, with relatively little effort, a number of features useful to other components across a broad range of the storage stack, including volume managers (enabling simple and efficient volume snapshots), filesystems (easily-integrated file snapshots), and transactional storage systems such as relational databases (allowing transactional updates without the double-write penalty). All measurements reported in this section were performed on an HP DL380p Gen8 server with two six-core (12-thread) 2.5GHz Intel Xeon processors and a 785GB Fusion-io ioDrive2, running Linux 3.4.

3.7.1 Snapshots

Snapshots are an important feature of modern storage systems and have been implemented at different layers of the storage stack from filesystems to block devices [131]. ANViL easily supports snapshots at multiple layers; here we demonstrate file- and volume-level snapshots.

File Snapshots

File-level snapshots enable applications to checkpoint the state of individual files at arbitrary points in time, but are only supported by a few recent filesystems [93]. Many widely-used filesystems, such as ext4 [94] and xfs [134], do not offer file-level snapshots, due to the significant design and implementation complexity it would incur.

ANViL enables filesystems to support file-level snapshots with minimal implementation effort and no changes to any internal data structures. Snapshotting individual files is simplified with the range clone operation, as all the filesystem needs to do is allocate a region of the address space of its backing block device (a region of ANViL logical address space) and issue a range operation to clone the address mappings from the existing file into the newly-allocated space [62]. The semantics this provides at the filesystem level are essentially identical to those of filesystems with built-in snapshot support, such as btrfs and zfs – the contents of the two files are identical and share the same physical storage, but writes to either one are transparently redirected to new physical locations without disturbing the contents of the other. The only slight semantic difference is in apparent space consumption – even though the data of the cloned files is in fact stored in the same physical blocks, as seen by the filesystem (and tools

examining it, such as `du` and `df`) the files appear to consume space independently of each other, since the filesystem cannot directly observe them being mapped to the same space in ANViL's backing device. However, given ANViL's inherent nature as a thin-provisioned storage system, some differences from "normal" intuitive space accounting are expected. If such *apparent* (if not actual) space consumption were to be problematic, for example in a filesystem making heavy use of file cloning, the natural solution would be to simply expand the allocation of ANViL's thin-provisioned logical space dedicated to the filesystem, allowing the filesystem plenty of apparent (logical) space in which to operate.

With just a few hundred lines of code, we have added an `ioctl` to ext4 to allow a zero-copy implementation of the `cp` command, providing an efficient (in both space and time) file-snapshot operation. Figure 3.4 shows, for varying file sizes, the time taken to copy a file using the standard, unmodified `cp` on an ext4 filesystem mounted on an ANViL device in comparison to the time taken to copy the file using our special range-clone `ioctl`. Unsurprisingly, the range-clone based file copy is dramatically faster than the conventional read-and-write approach used by the unmodified `cp`, copying larger files in orders of magnitude less time. Additionally, unlike standard `cp`, the range-clone based implementation shares

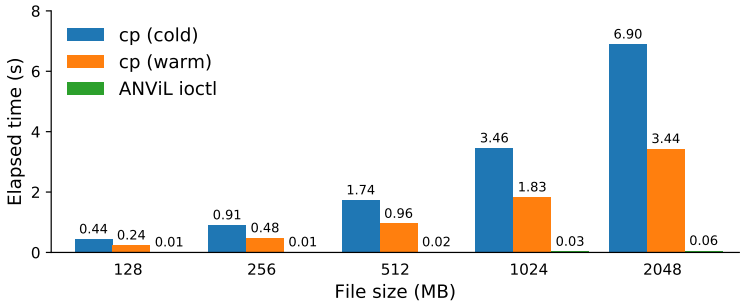


Figure 3.4: Time to copy files of various sizes via standard `cp` with both a cold and a warm page cache, and using a special ANViL `ioctl` in our modified version of ext4.

physical space between copies, making it also vastly more storage efficient.

Volume Snapshots

Volume snapshots are similar to file snapshots, but even simpler to implement. We merely identify the range of blocks that represent a volume and clone it into a new range of logical address space, to which a volume manager can then provide access as an independent volume.

Volume snapshots via range-clones offer much better performance than the snapshot facilities offered by some existing systems, such as Linux's built-in volume manager, LVM. LVM

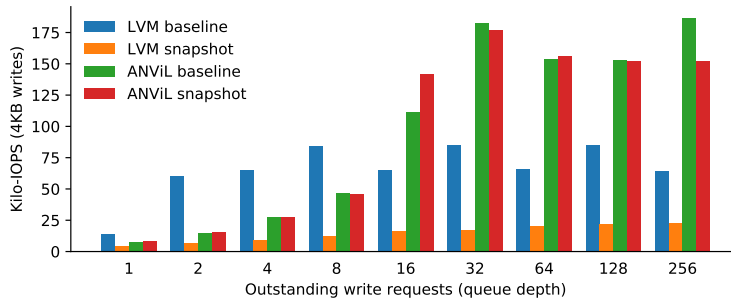


Figure 3.5: Random write IOPS on ANViL and LVM, both in isolation and with a recently-activated snapshot. The baseline bars illustrate ANViL's raw I/O performance. Its relatively low performance at small queue depths is due to the overhead incurred by its metadata updates.

snapshots are slow (somewhat notoriously so), because they operate via copy-on-write of large extents of data (2MiB by default) for each extent that is written to in the original volume from which the snapshot was taken. To quantify this, we measure the performance of random writes at varying queue depths on an LVM volume and on ANViL, both with and without a recently-created snapshot. In Figure 3.5, we see that while the LVM volume suffers a dramatic performance hit when a snapshot is active, ANViL sees little change in performance, since it instead uses its innate redirect-on-write mechanism. While this experiment was performed at a rela-

tively low space utilization level and hence does not reflect the performance impact of garbage collection, the performance impact of GC activity is in the contention for physical I/O bandwidth it adds, and hence would be expected to affect both the baseline and snapshot cases equally.

3.7.2 Deduplication

Data deduplication is often employed to eliminate data redundancy and better utilize storage capacity by identifying pieces of identical data and collapsing them together to share the same physical space. Deduplication can be implemented easily using a range clone operation. As with snapshots, deduplication can be performed at different layers of the storage stack. Here we show how block-level deduplication can be easily supported by a filesystem running on top of an ANViL device.

Extending the same `ioctl` used to implement file snapshots, we add an optional flag to specify that the filesystem should, as a single atomic operation, read the two indicated file ranges and then conditionally perform a range clone if and only if they contain identical data. This operation provides a base primitive that can be used as the underlying mechanism for a userspace deduplication tool, with the atomicity necessary to allow it to operate safely in the presence of possible

concurrent file modifications. Without this locking it would risk losing data written to files in a time-of-check-to-time-of-use race between the deduplicator observing that two block ranges are identical (the check) and then actually performing the range-copy operation (the use). While the simplistic proof-of-concept deduplication system we have is unable to detect previously-deduplicated blocks and avoid re-processing them, the underlying mechanism could be employed by a more sophisticated offline deduplicator without this drawback (or even, with appropriate plumbing, an online one).

3.7.3 Single-Write Journaling

Journaling is widely used to provide atomicity to multi-block updates and thus ensure the consistency of metadata (and sometimes data) in systems such as databases and filesystems. Such techniques are required because storage devices typically do not provide any atomicity primitives beyond the all-or-nothing behavior guaranteed for a single-block write. Unfortunately, journaling causes each journaled update to be performed twice: once to the journal region and then to the final "home" location of the data. In the event of a failure, such as a system crash or power loss, updates that have been committed to the journal are replayed at recovery time and applied the the corresponding primary persistent data

structures, and uncommitted updates are simply discarded. ANViL, however, can leverage its redirect-on-write nature and internal metadata management to support a multi-block atomic write operation, even across discontiguous regions of the logical address space. With this capability, we can avoid the double-write penalty of journaling and thus improve both performance and the lifespan of the flash device.

By making a relatively small modification to a journaling filesystem, we can use a vectored atomic range move operation to achieve this optimization. When the filesystem would write the commit block for a journal transaction, it instead issues a single vector of range moves to atomically relocate all metadata (and/or data) blocks in the journal transaction to their "home" locations in the main filesystem. Figure 3.6 illustrates an atomic commit operation via range moves. This approach is similar to Choi *et al.*'s JFTL [31], though unlike JFTL the much more general framework provided by ANViL is not tailored specifically to journaling filesystems.

Using range moves in this way obviates the need for a second write to copy each block to its primary location, since the range move has already materialized them there, eliminating the double-write penalty inherent to conventional journaling. This technique is equally applicable to metadata journaling and full data journaling; with the latter this means that a

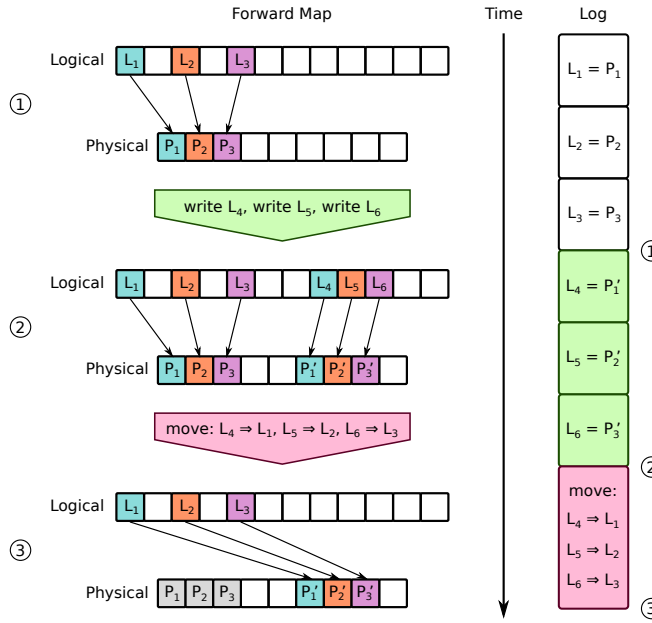


Figure 3.6: Transactions via address remapping. By using an application-managed scratch area, atomic transactional updates can be implemented using range operations. At ① the system is in its initial pre-transaction state, with logical blocks L_1, L_2, and L_3 each mapped to blocks containing the initial versions of the relevant data. Between ① and ②, new versions of these blocks are written out to logical addresses in a temporary scratch area (L_4, L_5, and L_6). Note that these intermediate writes do not have to be performed atomically. Once the all writes to the temporary locations in the scratch area have completed, a single atomic vectored range-move operation remaps the new blocks at L_4, L_5, and L_6 to L_1, L_2, and L_3, respectively, transitioning the system into state ③, at which point the transaction is fully committed. The recovery protocol in the event of a mid-transaction failure is simply to discard the scratch area.

filesystem can achieve the stronger consistency properties offered by data journaling without paying the penalty of the doubling of write traffic incurred by journaling without range moves. By halving the amount of data written to the backing device, the lifespan of flash storage chips is also increased due to the smaller number of program/erase cycles incurred.

Implementing transactional commits via range-move operations also obviates the need for any journal recovery at mount time, since any transaction that has committed will need no further processing or I/O, and any transaction in the journal that has not completed should not be replayed anyway (for consistency reasons). This simplification would allow the elimination of over 700 lines of relatively intricate journal-recovery code from the jbd2 codebase that provides ext4's journaling machinery.

In effect, this approach to atomicity simply exposes to the application (the filesystem, in this case) the internal operations necessary to stitch together a vectored atomic write operation from more primitive operations: the application writes its buffers to a region of scratch space (the journal), and then, once all of the writes have completed, issues a single vectored atomic range move to put each block in its desired final location.

We have implemented single-write journaling in ext4's

jbd2 journaling layer; it took approximately 100 lines of new code and allowed the removal of over 900 lines of existing commit and recovery code. Figure 3.7 shows the performance results for write throughput in data journaling mode of a process writing to a file in varying chunk sizes and calling `fdatasync` after each write. In all cases, ext4a (our modified, ANViL-optimized version of ext4) achieves substantially higher throughput than the baseline ext4 filesystem.

At small write sizes the relative performance advantage of ext4a is larger, because in addition to eliminating the double-write of file data, the recovery-free nature of single-write journaling also obviates the need for writing the start and commit blocks of each journal transaction; for small transactions the savings from this are proportionally larger. At larger write sizes, the reason that the performance gain is less than the doubling that might be expected (due to halving the amount of data written) is that despite consisting purely of synchronous file writes, the workload is actually insufficiently I/O-bound. The raw performance of the storage device is high enough that CPU activity in the filesystem consumes approximately 50% of the workload's execution time; jbd2's `kjournald` thread (which performs all journal writes) is incapable of keeping the device utilized, and its single-threadedness means that adding additional userspace

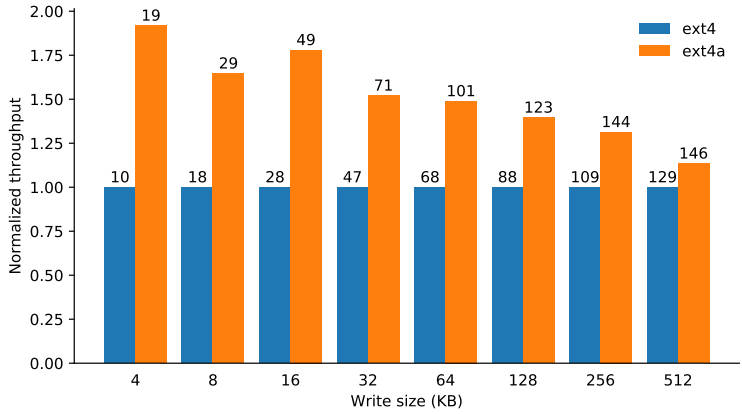


Figure 3.7: Data journaling write throughput with ANViL-optimized ext4a compared to unmodified ext4. Each bar is labeled with absolute write bandwidth (MiB/second).

I/O threads to the workload does little to increase device I/O bandwidth utilization.

The mechanism underlying single-write journaling could be more generally applied to most forms of write-ahead logging, such as that employed by relational database management systems [108].

3.8 GC Evaluation

ANViL is a complex system (approximately 100K lines of code), with garbage collection contributing significantly to its complexity. In this section, we examine the GC specifically. In doing so, we dissect the various aspects that contribute to the cost of reclaiming a segment and evaluate how the GC scales with large storage capacities and numbers of references to the same data. The experiments in this section were performed using SuSE Linux Enterprise Server SP2 with a 3.0 Linux kernel, running on an HP DL380 server with 64GiB of RAM, two 6-core (12-thread) Intel Xeon processors, and a 1.2 TB SanDisk ioMemory PCIe flash drive as ANViL's backing storage.

3.8.1 Garbage Collection in Action

Here we demonstrate the basic operation of the ANViL GC, including how it ramps up its activity as device utilization increases. To illustrate this behavior, we artificially reduce the capacity of the backing device to 320 GB. We ran a workload using `fio` [14] with 32 threads each writing 10GiB of data in 512B blocks, with an overwrite ratio of 50%. Figure 3.8 shows the progress of the system over time.

Initially, when device utilization is low, the GC is granted

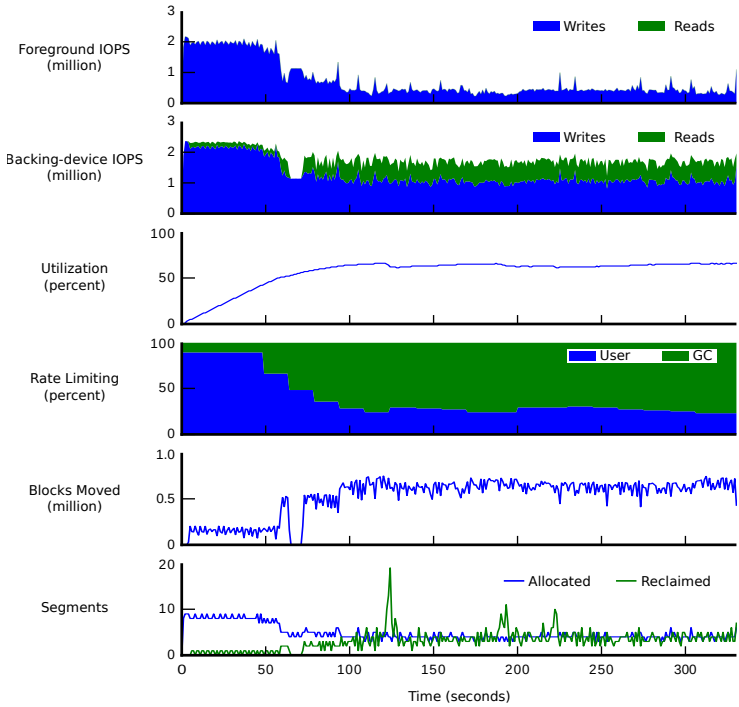


Figure 3.8: Steady-state GC activity. This figure shows the operation of the GC under a steady, intense, random-write workload starting from a freshly-initialized (empty) state. As the overall space utilization grows, the rate limiter allocates an increasing fraction of the backing device's I/O bandwidth to garbage collection, eventually reaching a stable equilibrium at which the garbage collector reclaims segments at roughly the same rate as they are allocated to accommodate incoming write requests.

Data	Mappings	GC Time (seconds)
128M	3,712	0.3
8G	266,952	28.3
64G	2,152,134	179.4
128G	4,037,073	363.7

Figure 3.9: GC capacity scaling. We populate the device with some data and alter the GC to clean segments even though they contain only live data. The cost in time and mappings scanned thus represents the time spent by the GC in moving all of the data that was originally written.

only a small fraction of the available I/O bandwidth and hence foreground traffic proceeds at essentially full throttle. Once space utilization crosses 50%, the rate limiter begins increasing the GC's bandwidth allocation and the cleaner starts performing significant I/O to move data blocks out of reclaimed segments, as can be seen at approximately 60 seconds. At approximately 120 seconds, we observe that the system as a whole has reached a sustainable steady state, with the GC keeping up with the incoming write stream, as evidenced by the roughly equal rates of segment allocation and reclamation.

3.8.2 GC Capacity Scaling

It is important for the GC to scale up gracefully when ANViL is used to store large volumes of data. To evaluate this, we measure the time taken by the GC to reclaim the valid data from a set of segments. In these experiments, we issue 4KiB sequential writes to the device and allow the GC to start processing after all the writes complete. We modify the scanner's candidate-selection code to make it reclaim any segment, regardless of its utilization. In this particular experiment, as we have no overwrites in our initial workload, each segment contains entirely valid data. Figure 3.9 shows the number of mappings traversed during the scan and the time taken by the GC to scan and clean all the written segments; the scan time increases roughly linearly with the quantity of data.

The scanner's throughput can be increased substantially by parallelizing the scan across multiple threads. To measure this, we wrote 128 MiB of data to ANViL (one segment's worth) and created 10,000 snapshots of it (to populate the forward map with as many mappings as would be used for 1.28TiB of non-snapshotted data). We then measured the performance of the scanner (the rate at which it traverses the forward map) with varying numbers of threads. The results in Figure 3.10 show that the multithreaded scanner is highly

scalable, allowing ANViL to scale to large numbers of data references by making use of the plentiful CPU cores of modern systems.

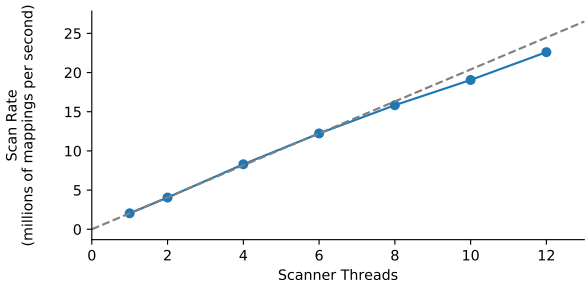


Figure 3.10: Scanner scalability. This figure illustrates the scalability of the GC's multithreaded scanning, showing scanning performance at varying thread counts. The scanner achieves near-linear scaling up to 12 threads (the number of CPU cores on the test system). The dashed line represents perfect linear scaling extrapolated from the measured performance of a single thread.

3.9 Conclusion

The case studies presented in this chapter demonstrate that with a simple but powerful block-address remapping mechanism, a single log-structured storage layer can provide appli-

cations and filesystems above it with both high performance and a flexible storage substrate. The clone, move, and delete range operations with which ANViL augments the conventional block-I/O interface provide a great deal of added utility to higher-level software while remaining simple to integrate.

We have demonstrated how these operations can be used to enable, with relatively minor changes to an existing codebase, efficient implementations of an assortment of storage features. The clone operation can create a volume snapshot without compromising I/O performance, deduplicate file contents, or implement an accelerated file-copy operation that can run over 100 times faster than the standard `cp` command. The vectored move operation can enable single-write transactional commits, which we have shown can provide as much as a 90% performance increase in write throughput when applied to ext4's data journaling mode (while also simplifying the existing code).

Beyond its immediate utility to applications, however, ANViL is designed for modern hardware. Its GC's multi-threaded scanner takes advantage of plentiful CPU cores to improve its throughput and decrease the amount of time required to scan through its logical address space, and the scanner's performance scales up gracefully as more CPUs are added. ANViL's I/O patterns harmonize well with the

characteristics of flash storage devices. All writes issued to its backing storage are gathered together into large, sequential streams. Its GC also utilizes the available internal I/O parallelism of high-performance SSDs by issuing batches of concurrent read requests when internally relocating data. Its avoidance of small random writes in combination with the GC's cleaner regularly issuing large TRIM requests help to ease the workload on the internal GCs of the underlying flash devices, keeping them running smoothly. ANViL is thus a storage virtualization system well-suited to the types of hardware that have become prevalent in the landscape of modern computing.

4

Cache-Conscious Filesystems for Low-Latency Storage

The filesystem is a venerable abstraction that has endured over decades of development and numerous generations of hardware and software. The familiar Unix-style structure of variable-size files containing arbitrary byte arrays organized in a hierarchical directory tree has now existed for over half of the entire history of digital electronic computing as we know it today [118]. Its model is highly general while still being conceptually simple – low-level enough to serve as a substrate for arbitrary storage in programmatic use, high-level enough for direct access by human users to be comfortable. The filesystem abstraction has thus become deeply entrenched in modern computing systems, and appears highly unlikely to

be replaced anytime in the near future.

While the *interface* of the filesystem abstraction has endured, however, the methods with which it is implemented have evolved considerably, with myriad designs for filesystem internals proposed and implemented over the years [22, 62, 79, 96, 118, 121, 134]. Though these implementations often differ from one another quite radically, a common aspect is the nature of the hardware for which they are designed: block-based storage devices accessed via (to varying degrees) relatively high-latency asynchronous operations. Now, however, an entirely new class of persistent storage hardware, not matching this description at all, appears poised to present the next major jump forward in storage technology: NVM (nonvolatile memory) devices are byte-addressable and provide low-latency access via regular load and store CPU instructions.

The arrival of this new and dissimilar technology thus raises the question of how best to implement the traditional filesystem abstraction on top of it. Given its much lower access latencies, the CPU utilization of the software providing this abstraction becomes a much more important factor than it has been with slower block storage hardware. In this chapter we examine one particularly critical aspect of CPU performance, cache behavior, in the context of filesystem design for NVM hardware.

4.1 Introduction

Storage device speeds have increased considerably with the widespread adoption of flash in system that previously had employed hard disk drives [71, 79]. With the increasing availability of non-volatile memory (NVM) technologies [45, 54, 130], systems with persistent storage accessible with DRAM-like latencies may soon be widespread. With these dramatic improvements in the performance of storage hardware, the overhead incurred by the software managing it becomes more and more significant and storage-intensive applications that were previously I/O-bound become increasingly CPU-bound. This transition has led to research efforts into techniques like kernel-bypass filesystems [27, 111, 142–144] and in-device filesystems [74].

One of the most important factors in the CPU performance of a workload is its hit rate in the CPU cache [7, 44, 78], a hardware resource shared by both the application and the operating system's storage stack. This sharing means that in addition to the performance of filesystem code itself, the design and implementation of performance-conscious filesystems should also give consideration to the effects of cache pollution – that performing filesystem operations perturbs the delicate cache state needed to achieve good performance in executing non-filesystem code.

However, filesystem research thus far has spent little effort on this facet of the storage stack. Software design decisions both small and large, as well as phenomena such as code alignment that are not typically consciously decided by software developers (but can be controlled by a programmer who is aware of them), can play a significant role in a filesystem's cache behavior.

In order to examine and experiment with its impact on application performance, in this chapter we study the cache footprints and access patterns of different Linux filesystems. We then develop an experimental filesystem, DenseFS, with the explicit aim of having a compact cache footprint, and evaluate the performance benefits of the reduced pollution of application cache state that this smaller footprint provides. With targeted microbenchmarking we find that in comparison to an array of existing Linux filesystems, DenseFS can dramatically reduce the performance impact of the cache pollution caused by filesystem operations, in some cases reducing a 150% overhead to merely 20%. Using a real-world program, we find that using DenseFS in place of other existing filesystems can achieve a 37-65× reduction in L1 instruction cache misses, providing a 13% to 18% improvement in user-mode CPU performance.

The remainder of this chapter is organized as follows.

In Section 4.2 we investigate the cache behavior of existing Linux filesystems. Section 4.3 presents the design and implementation of DenseFS, as well as a more integrated second-generation version of it. Section 4.4 we evaluate the performance of both versions of DenseFS in comparison to other filesystems. Finally, Section 4.6 concludes.

4.2 Filesystem Cache Access Patterns

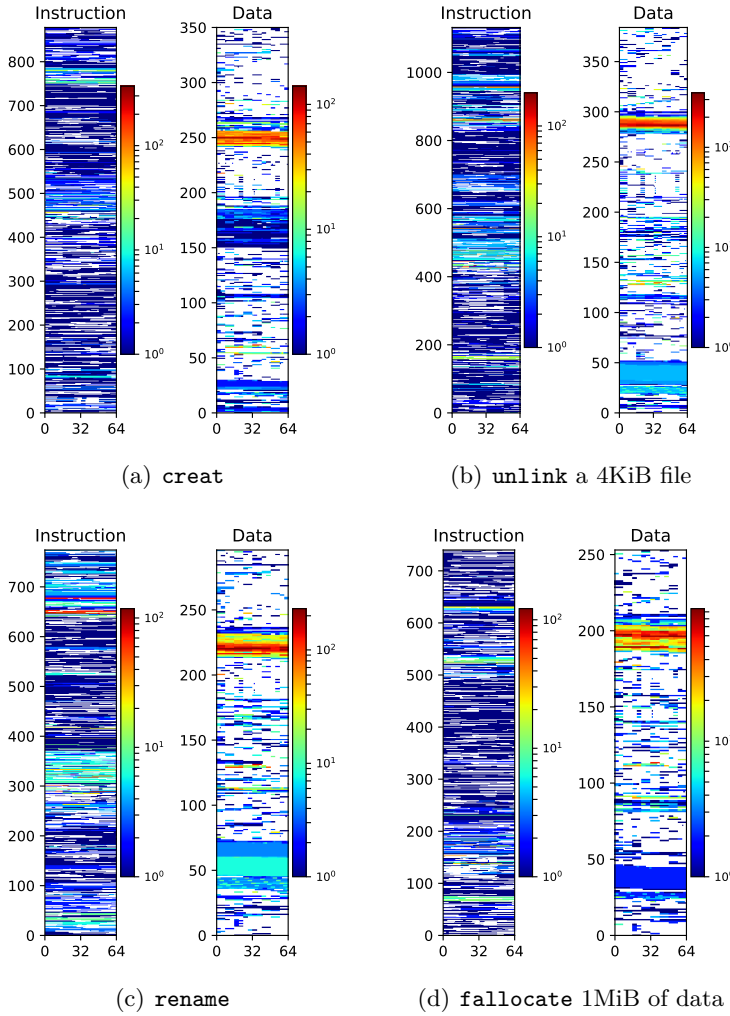
We begin with an investigation of the cache behavior of operations in existing Linux filesystems. We aim to determine, for both data and instruction accesses, the overall sizes of their cache footprints, how efficiently they utilize the cache (degree of reuse, whether bytes fetched into the cache go unaccessed and thus wasted), and what the main sources of their cache footprints are. We examine btrfs, ext4, f2fs, xfs, and tmpfs. The first four are the main persistent filesystems currently in widespread use (to varying degrees) on servers, desktops, and mobile devices running Linux. The final filesystem we analyze, tmpfs, is a non-persistent in-memory filesystem which has been discussed by Linux developers as a possible basis for NVM filesystem support [101].

By scripting `gdb` attached to the kernel running in a virtual machine, we collect instruction-level dynamic traces

of btrfs, ext4, f2fs, xfs, and tmpfs performing an assortment of metadata operations. We trace the entire kernel-mode execution of each system call, recording for each instruction its address and size, the addresses and sizes of any data memory accesses it performs, and the full symbolic stack backtrace (the function name, source file, and line number for each stack frame).

Our first analysis processes these traces by aggregating all instruction and data memory accesses at byte granularity and counting the number of times each individual byte is accessed. We continue along the path of prior research in using heatmaps for visualizing cache access patterns [33, 138, 157] with a special heatmap we term a *cachemap* (see Figures 4.1-4.4). Each row of cells in a cachemap represents a single cache line (64 bytes), with each cell representing one byte of memory. The vertical axis serves simply to order cache lines by virtual address, though it is not generally contiguous (only cache lines that were accessed at least once are shown). The color of each cell provides a log-scale indication of how many times that byte was accessed[1] throughout the entire trace (with white representing the special value zero).

[1]The program that generates these cachemaps also offers an interactive mode in which the user can click on a cell to see the full backtrace of every point at which that byte was accessed, making it easier to identify opportunities for potential optimizations.



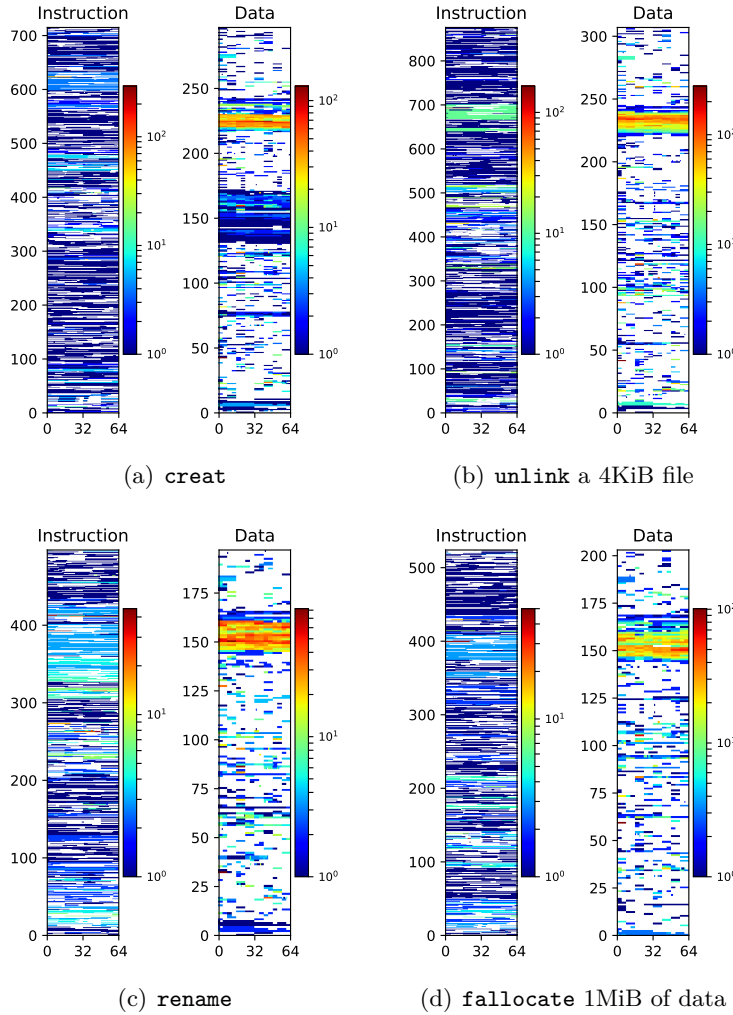
(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.1: Cachemaps of metadata operations on btrfs.



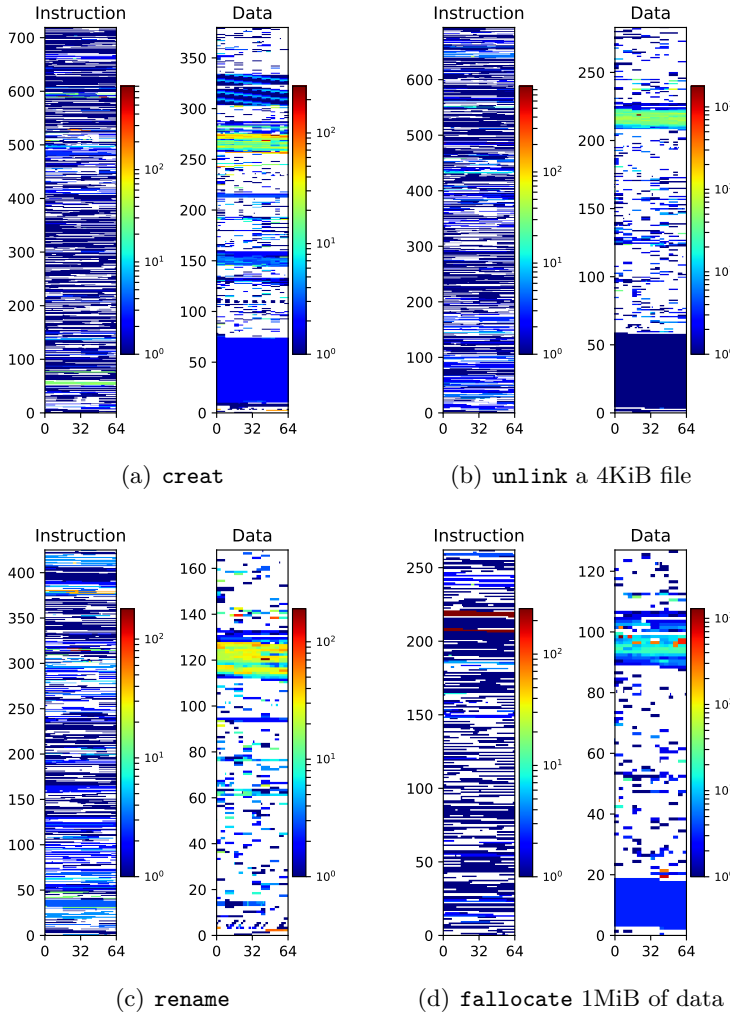
(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.2: Cachemaps of metadata operations on ext4.



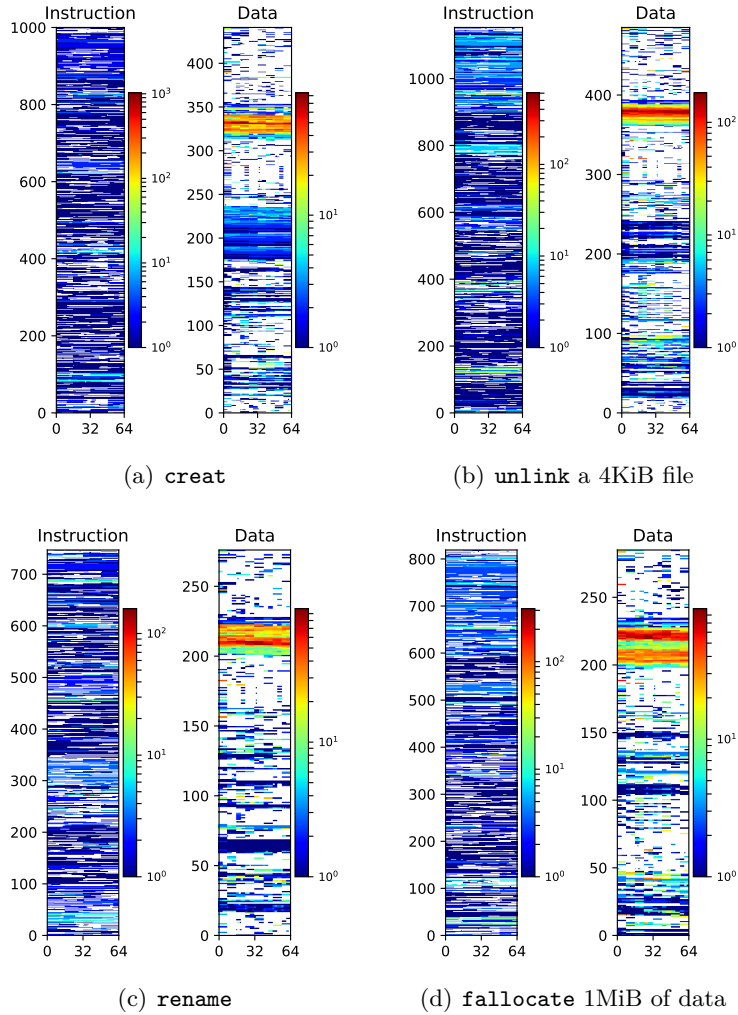
(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.3: Cachemaps of metadata operations on f2fs.



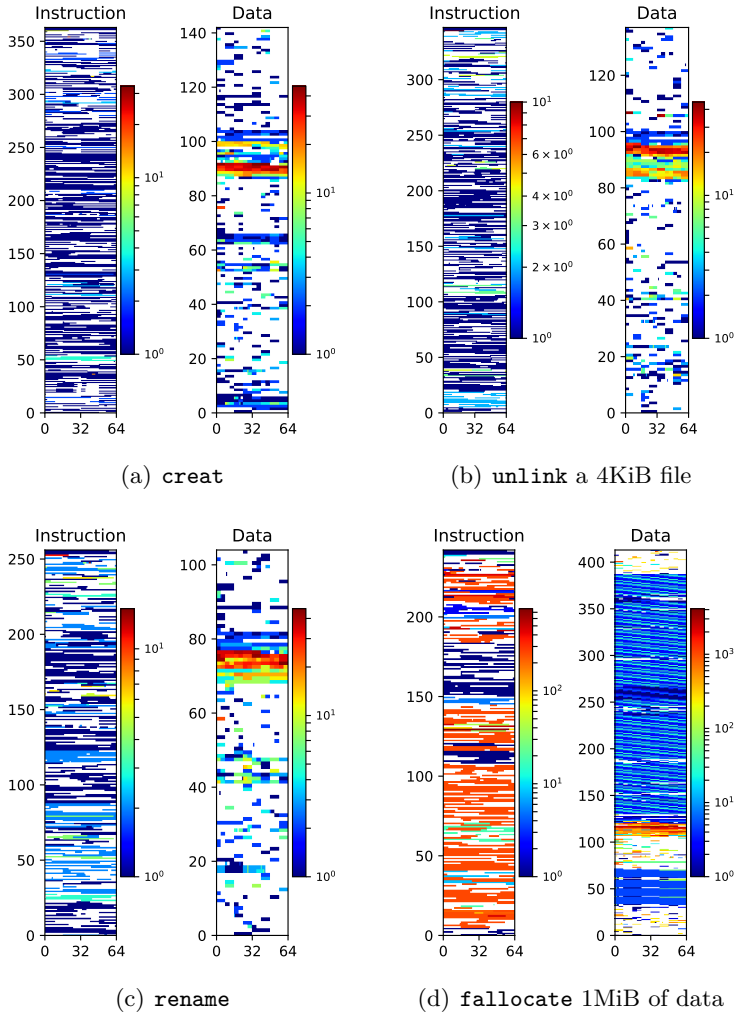
(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.4: Cachemaps of metadata operations on xfs.



(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.5: Cachemaps of metadata operations on tmpfs.

We begin at a high level: looking at the sizes of overall cache footprints, it is clear that all of these operations will significantly perturb the first-level caches, if not displace their contents entirely. Current generations of x86 processors have L1 instruction and data caches of 32KB each (512 64-byte cache lines). Of the twenty operations in our cachemaps, thirteen show instruction cache footprints that exceed the size of the L1 cache. The data cache footprints are generally roughly half the size of the code footprints, with twelve operations exhibiting data footprints over half the size of the L1 cache. Executing operations like these will thus significantly disturb warm L1 cache state built up by an application during its execution, degrading its performance after the system call completes until the application's working set can be brought back into the cache.

We see in these cachemaps that many data cache accesses are relatively wasteful in that they drag an entire 64-byte line into the cache (displacing another one, which may have contained useful application working-set data) only to provide a small handful of bytes, often for a single, isolated memory access. Accesses of this sort exhibit neither the spatial nor the temporal locality for which caches are optimized, and hence make poor use of them.

The instruction access patterns shown in our cachemaps

indicate a different inefficiency in their cache utilization. Instruction fetches, due to execution being inherently sequential by default (in the absence of branches), are somewhat less wasteful in that a smaller number of bytes in each cache line go unused on average. However, despite this spatial locality, the prevalence of dark blue cells in the instruction cachemaps indicate that there is relatively little temporal locality (reuse of already-cached instructions); given the larger size of the instruction cache footprint this is still not a particularly effective use of hardware resources.

One of the more eye-catching features of these cachemaps is that every data-cache map shows a densely- and heavily-accessed region of perhaps ten to twenty cache lines that stands out from everything else: this is simply the C execution stack, which exhibits cache-friendly behavior with its high degree of spatial and temporal locality. While it may in some cases be possible to reduce the stack footprint of a given sequence of code, the reuse of the same stack space by different function calls means that reducing the *overall* stack footprint is unlikely to happen anywhere but at the outermost (leaf) levels of the call tree, and only when the stack is at its deepest, making the stack an unpromising area for efforts toward cache-footprint optimization.

In many data cachemaps (for example, around cache line

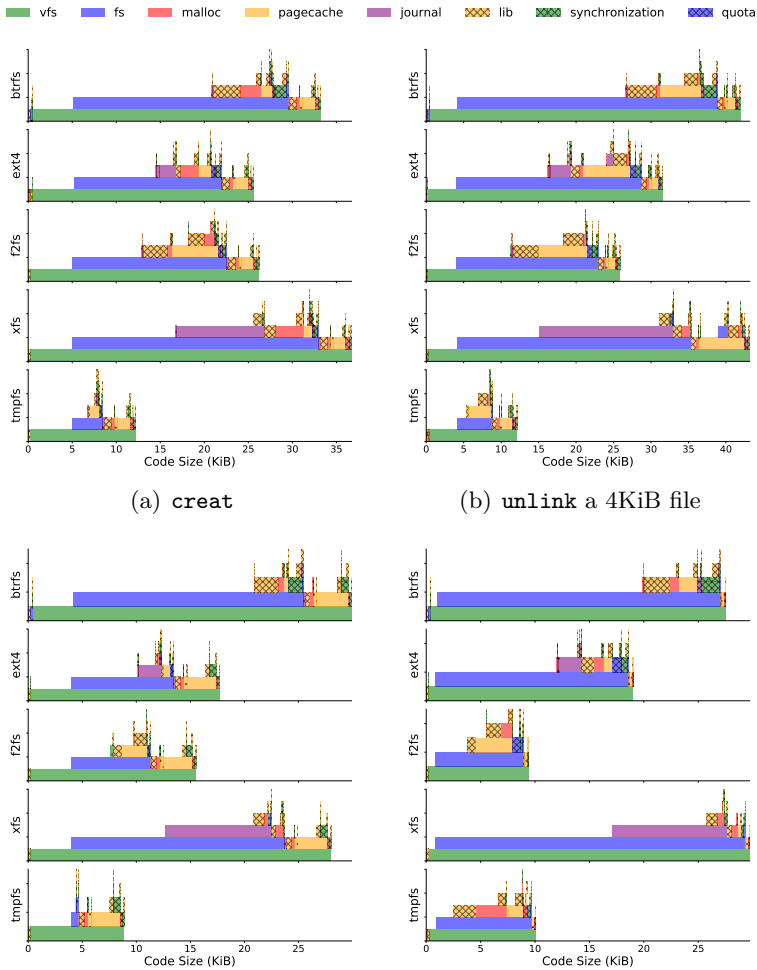
225 of Figure 4.1(b), cache line 180 of Figure 4.2(a), and cache line 300 of Figure 4.4(b)) we see regions of similarly-patterned accesses to a number of cache lines. This phenomenon occurs as a result of the memory layout of common data structures, such as `struct inode` and `struct kmem_cache`. Certain code paths will access certain specific subsets of the members of common structs such as these, often leaving other members untouched. The clear visualization of these patterns provided by our cachemaps can make it easy to identify opportunities for memory layout micro-optimizations, such as rearranging the layout of `struct kmem_cache` such that the members that are needed by the performance-critical common-case allocation path are grouped into a single cache line instead of spanning two lines.

A particular case that stands out visually is tmpfs's `fallocate` operation, shown in Figure 4.5(d). Whereas the other filesystems examined have extent-based data structures that allow them to efficiently allocate large, contiguous regions of space, tmpfs operates only on individual pages (4KiB each). When allocating a large amount of space, it thus needs to execute the same page-allocation code many times (allocating single 4KiB pages 256 times to satisfy a 1MiB `fallocate` request), leading to a much higher degree of instruction reuse, as well as the dense, patterned data accesses it exhibits as it ac-

cesses members of the data structures representing the pages it allocates.

Due to their larger sizes, we focus first on optimizing instruction cache footprint. We wish to gain a high-level understanding of what software components are the main contributors to the overall size of that footprint so as to guide our efforts to reduce it. The low-level nature of instruction traces and the cachemaps we have examined thus far, however, makes it difficult to discern the major sources of instruction cache footprint. In order to look at our trace data from a vantage point more appropriate for this analysis, we condense our instruction traces into *coarse-grained stack traces*, or *cgstacks* and visualize each of them in the form of a flame graph [51].

A cgstack is a simplified view of the stack backtrace of a given instruction. Given an instruction's stack backtrace, we transform it into a cgstack by mapping each frame, progressing from callers to callees, to one of a set of designated code categories based on the file in which that function is defined (for example, functions in `mm/slab.c` are mapped to the "malloc" category, while `fs/file.c` is mapped to the "vfs" category). If the category classification of a given stack frame has not yet been seen in the corresponding cgstack thus far, that category is then added to the top of the cgstack. The re-



(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.6: Cgstack flame graphs of the components contributing to the code footprints of Linux filesystems.

sult is effectively a high-level statement about the provenance of each instruction. For example, for a given instruction that statement may be that the instruction's presence in the trace is attributable to page cache code called by VFS code. After transforming each instruction's stack trace into a cgstack in this way, we then weight each cgstack by the size of the instruction and aggregate the data together, producing a flame graph for each trace (see Figure 4.6).

Of the two journaling filesystems examined, we see that in all cases xfs's journaling code occupies a much larger footprint than that of the corresponding ext4 code (4.8-5.2×). This disparity aligns with expectations, as ext4's jbd2 layer performs simpler physical journaling [148], whereas xfs employs a more complex hybrid logical/physical journaling scheme [30].

In almost all cases the footprint of the VFS code (and other non-filesystem-specific code transitively executed by it) is quite uniform, with overall footprint differences between filesystems stemming entirely from filesystem-specific code (the "fs" category). The one evident exception is the `unlink` operation on xfs. This difference is simply an artifact of the fact that xfs is the only filesystem of these five that does not define its own `evict_inode` operation, instead using the default implementation provided by the vfs. However, the `evict_inode` operations employed by btrfs, ext4, and

f2fs all (amidst other, filesystem-specific work) call the same `truncate_inode_pages_final` function called by the default VFS code path that xfs uses, so the code that is ultimately executed is largely similar, despite being invoked via a different path.

Over a quarter of tmpfs's code footprint (2.8KiB of its 10KiB) for the `fallocate` operation comes from memory-allocation code (the "malloc" category). This composition is a natural result of tmpfs being a memory-based filesystem; allocating space for file data is thus internally a memory-allocation operation. (This is another manifestation of the same phenomenon described earlier in our cachemap analysis, in which tmpfs's memory access patterns for `fallocate` look markedly different from any other operation.)

While these cgstack flame graphs do show variations between different filesystems, they also make it clear that common, non-filesystem-specific infrastructure such as the VFS and page cache play a significant role in overall code footprint. Armed with this knowledge, we set out to construct a new filesystem with the aim of maximizing cache density, in part by keeping it disentangled from the conventional filesystem framework. The resulting filesystem is called DenseFS, and is detailed in Section 4.3.

4.3 DenseFS

DenseFS is a small in-memory Linux filesystem implemented in approximately 2500 lines of code. Our initial aim with DenseFS is not to provide a full-featured, robust, "real" filesystem, but rather an experimental system to explore the potential performance benefits (both in the speed of its own execution and in its impact on the user-mode performance of applications using it) of a filesystem with a greatly reduced cache footprint, even if that comes at some costs in practicality and ease of use by applications. It has not been optimized for scalability and lacks a number of features normally expected of any modern filesystem, such as crash-consistency, symlinks, and `mmap` support.

Given the results of our analysis in Section 4.2 showing that the VFS and page cache code are significant contributors to the large code footprints of existing filesystems, DenseFS is not integrated into the "normal" Linux VFS layer and does not use its page cache. This design choice is at the root of its primary practical difficulty: the standard file-access system calls (`open`, `read`, `unlink`, etc.) cannot be used to access it. Instead, it offers its own parallel set of system calls (`dfs_open`, `dfs_read`, `dfs_rename`, and so forth) with the same arguments, but which operate on files in the DenseFS namespace. DenseFS file descriptors are distinct from (and not

interchangeable with) normal file descriptors, but otherwise operate similarly. Alongside its existing file descriptor table and working directory, each process thus gains a separate DenseFS file descriptor table and DenseFS working directory.

Within its set of special system calls, however, DenseFS has familiar features. Directory entries, inodes, and a superblock are represented with C `structs`, with pointers linking them together in the same overall structure as is found in most Unix-style filesystems. These structs are allocated in memory, but instead of using the general-purpose in-kernel memory allocation routines (Linux's `kmalloc` family of calls), it instead performs one large allocation for the entire (fixed) capacity of the filesystem when it is mounted and then allocates its own internal structures within that region of memory (mimicking what would be done in a true NVM filesystem).

4.3.1 Data Cache Compaction

In keeping with DenseFS's aims of being compact, some familiar structures are implemented differently than in conventional filesystems, in particular its inode. A straightforward inode structure for an in-memory filesystem like DenseFS might closely resemble the `stat` struct used in the standard `stat` system call, and indeed this was our initial starting point with DenseFS. With a few additional fields needed internally

(a spinlock, a reference count for open files, and a union of pointers for directory entries and file data), this simple implementation, however, yields a 112-byte inode – larger than desired for a cache-dense filesystem. With that as a starting point we made a series of optimizations to reduce the size of the DenseFS inode structure.

Fewer, smaller timestamps: The `stat` struct uses the bulky 16-byte `struct timespec` (with separate second and nanosecond fields) to represent the file's `atime`, `mtime`, and `ctime` timestamps. We start by simply replacing these with the Linux kernel's internal 8-byte `ktime_t` (a single nanosecond value), and removing the `atime` member entirely, since access times are rarely actually used by applications and hence filesystems are frequently mounted with the `noatime` option anyway. This change saves 32 bytes by reducing the space spent on timestamps from 48 bytes to 16, with only a slight compromise in functionality.

Zero-byte inode numbers: Inode numbers are also relatively little-used, though unlike the inode's `atime` the only information they encode is a unique identifier, which can thus be removed without any compromise in functionality or semantics. Instead of storing an inode number in each inode, DenseFS's `stat` call instead populates the `st_ino` field with

a value derived from the in-memory address of the inode itself. In order to allow for these synthetic inode numbers to remain persistent (were DenseFS operating on real nonvolatile memory), we subtract the base address of the DenseFS memory region to form an offset instead of a raw pointer value, and then XOR this offset with a secret key stored in the DenseFS superblock in order to avoid leaking potentially-sensitive metadata to userspace [36]. This change saves eight bytes in the DenseFS inode struct, with no sacrifices in functionality or performance.

Out-of-line metadata deduplication: This optimization is based on the observation that the user, group, and mode fields contain little entropy. Even in filesystems containing many millions of files, there may be only a few hundred unique combinations of these three fields, so encoding this near-duplicate information in every individual inode is an inefficient use of space. In DenseFS we thus compress this information by keeping a filesystem-wide table of `<uid, gid, mode>` tuples and replacing the corresponding three entries in the inode struct with a single 16-bit index into this table. By replacing three 32-bit fields with 16 bits, this optimization saves another 10 bytes, though it is a compromise in multiple ways.

While the `<uid, gid, mode>` metadata itself no longer takes up space in the inode, it is still just as large in the global

table, and accessing it there will still require bringing another line into the cache. However, many inode accesses (such data read and write operations via open file descriptors) simply do not need to use this metadata, so the cost of of accessing another cache line in the global table is not incurred. Additionally, operations that read this information from multiple inodes (such as a rename, or a path lookup traversing multiple levels in the directory hierarchy) will commonly access the same locations in the table for multiple inodes, reusing the same cache line instead of multiplying the cache-footprint overhead. And while DenseFS does not currently implement this, the entries in the table could also be organized for locality (for example, putting entries for the same user near each other) so that even operations that don't access exactly the same entry are likely to access ones in the same cache line.

This approach also imposes an additional performance cost on update operations, which now need to determine the correct index to use for a given `<uid, gid, mode>` combination, and in the uncommon case of setting one that does not already exist somewhere in the filesystem, add an entry for it to the global table. In the current implementation of DenseFS, determining the index for a given metadata tuple is done via a simple linear search; in a more production-ready implementation this could be optimized with a TLB-like cache

of recently-used entries, possibly in combination with a more sophisticated data structure.

Special-cased "." and ".." directory entries: We made one additional data cache optimization unrelated to the layout of the DenseFS inode struct itself that reduces the number of cache lines accessed during path lookups. The initial implementation treated each directory's "." and ".." entries no differently than any others; given the simple linear-search directories DenseFS employs, this incurs additional cache-line accesses to check them during lookup operations in each directory. To avoid these extra accesses we instead modified the directory-search code, adding an explicit special-case check for these names instead of actually materializing them in each directory's list of entries.

Results

With all of these optimizations applied and with some additional savings from reordering a few inode fields to eliminate padding bytes, we achieve an important goal: at 56 bytes, the DenseFS inode struct is now small enough to be contained entirely in a single cache line. The entire layout of the resulting inode structure is shown in Figure 4.7. For comparison, Figure 4.8 shows the sizes of the in-memory inode structures

```
struct densefs_inode {
        uint16_t nlink; /* 2 bytes */
        metaidx_t meta_idx; /* 2 bytes */
        refcount_t refcount; /* 4 bytes */
        off_t size; /* 8 bytes */
        ktime_t mtime; /* 8 bytes */
        ktime_t ctime; /* 8 bytes */
        spinlock_t lock; /* 4 bytes */
        /* 4-byte hole for alignment  */
        union {
                struct list_head {
                        struct list_head *next;
                        struct list_head *prev;
                } dirents;
                struct rb_root {
                        struct rb_node *rb_node;
                } chunks;
        } data; /* 16 bytes */
};
```

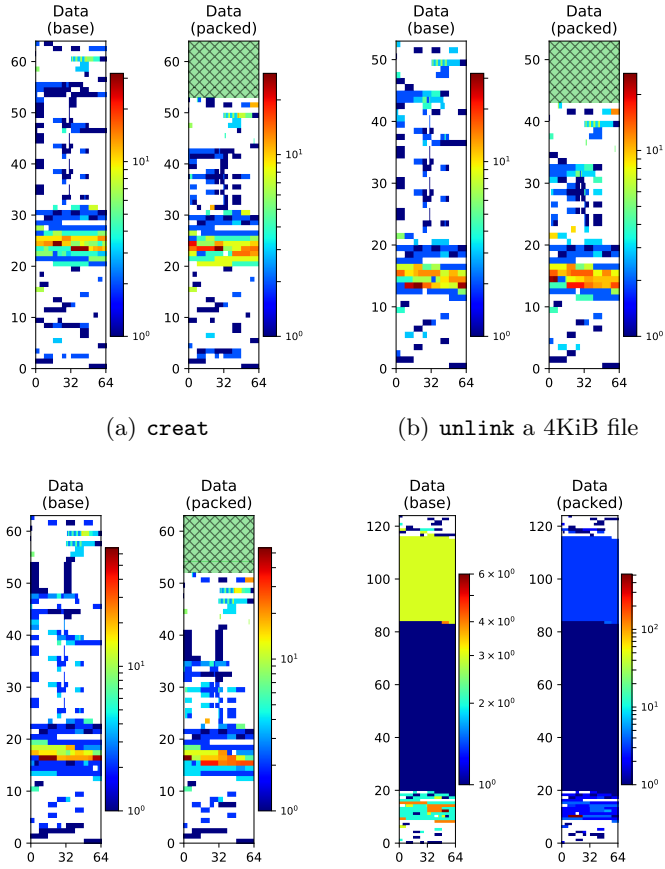
Figure 4.7: The 56-byte DenseFS inode structure. File data is stored in a red-black interval tree of contiguous extents (`data.chunks`); directory entries are kept in a simple linked list (`data.dirents`).

Filesystem	In-memory inode size (bytes)
btrfs	1,064
ext4	1,056
f2fs	928
xfs	920
tmpfs	680

Figure 4.8: In-memory inode sizes of Linux filesystems. 576 bytes of each inode is consumed by the generic VFS `struct inode` embedded within it.

for the five existing Linux filesystems we have evaluated. Over half of the size of these inode structures is due to the fact that each of them embeds an instance of the Linux VFS layer's 576-byte `struct inode`. Note, however, that while these structures are large, relatively few of their members are typically accessed by a given operation, so the effective impact on cache footprint is less dramatic than these raw `sizeof` numbers might imply.

The resulting decrease in data cache footprint can be seen in the cachemaps in Figure 4.9. The `creat`, `unlink`, and `rename` operations see reductions of 10 to 11 cache lines each, or 17-19%. These reductions are proportional to the depth of the file paths on which they operate; in these traces our benchmark program was configured to access files four



(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.9: Data cachemaps of DenseFS, before and after cache-compaction optimizations. The hatched green regions near the tops of the packed cachemaps indicate cache footprint eliminated by the optimizations described in Section 4.3.

directory levels deep in the filesystem. In combination with the starting directory (the DenseFS root) and the file itself, this implies accessing six inodes in total. The savings in the optimized version are thus proportional to two cache lines per level of the path lookup (one from the inode accessed at each level fitting in a single line instead of two, and one from avoiding accessing the `"."` and `".."` directory entries), though this is slightly offset by the additional access to the out-of-line metadata in the global `<uid, gid, mode>` table. `fallocate`, however, does not see any benefit from these optimizations, because it operates on a file descriptor instead of a path (and thus performs no directory lookups), so the one-line reduction from the compacted inode is balanced by the additional access to its external `<uid, gid, mode>` metadata.

4.3.2 Instruction Cache Compaction

To compact DenseFS's code footprint, we first traced its execution of various calls and produced corresponding cachemaps as in Section 4.2. Guided by these cachemaps, we then applied three varieties of manual adjustments to help the executed code fit into fewer cache lines.

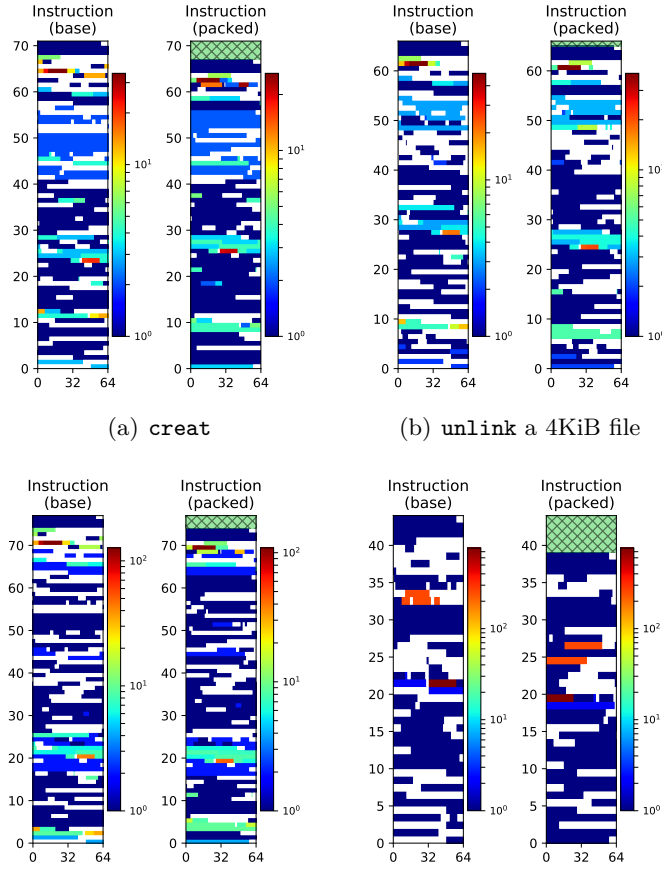
Function alignment: This optimization is the most frequently applicable and hence the most impactful technique.

The function `current_kernel_time64`, used in updating inode timestamps, provides an excellent example of it. The function's code is only 58 bytes long, short enough to fit in a single cache line, but its starting address is offset from the cache-line boundary such that it spills over into the next line, causing its execution to displace one more line than it truly requires. By annotating it to be aligned on a 64-byte boundary, we avoid this pitfall and keep it contained in a single cache line. It would be simple to use a compiler flag to apply this alignment constraint globally to all functions, but this is not necessarily always beneficial, as will be shown in our discussion of function ordering below.

Branch hinting: The opportunity for this optimization arises when the compiler arranges code suboptimally for a conditional such as an `if` block. Consider a simple example with an `if` block with a small body and no `else` clause. A straightforward compilation of the code might put the body of the `if` block "inline" with the surrounding code preceded by a conditional branch that skips over it when the condition is false. If the condition is rarely true, however, this results in wasted space in the instruction cache – the bytes for those instructions are brought into the cache alongside their neighboring instructions, but are never executed. If the bias of the condition is known, a more optimal compilation would

instead place the body of the `if` block in a relatively far-off location after the main "hot" body of the function and branch to it (and then back) in the unlikely case that its condition is true. By identifying occurrences like this (which are visible as small gaps of white in our cachemaps), we can sometimes add appropriate annotations to such `if` conditions and squeeze out a few more precious bytes of wasted cache space.

Function ordering: In one case we observed a cluster of three functions, one 30 bytes, one 37, and one 28 bytes (`strcpy`, `strcmp`, and `strlen`, respectively). Despite totaling only 95 bytes, they nevertheless spanned four cache lines – 256 bytes worth of space. One of the two extra "wasted" lines was due to suboptimal alignment of `strlen` causing its code to spill onto a second line, but even after addressing that the trio of string functions that should have fit easily in two lines still consumed three. Despite being defined in the same source file, their relatively distant locations within that file led to the corresponding layout in memory not condensing them together as would be desirable for compactness. In this case, cache-line-aligning all three functions individually would still reduce cache density by the same token – separating closely-related pieces of code. By simply reordering the functions to bring them together in the source file that defines them (`lib/string.c`), we were able to achieve the desired result of



(a) `creat` (b) `unlink` a 4KiB file

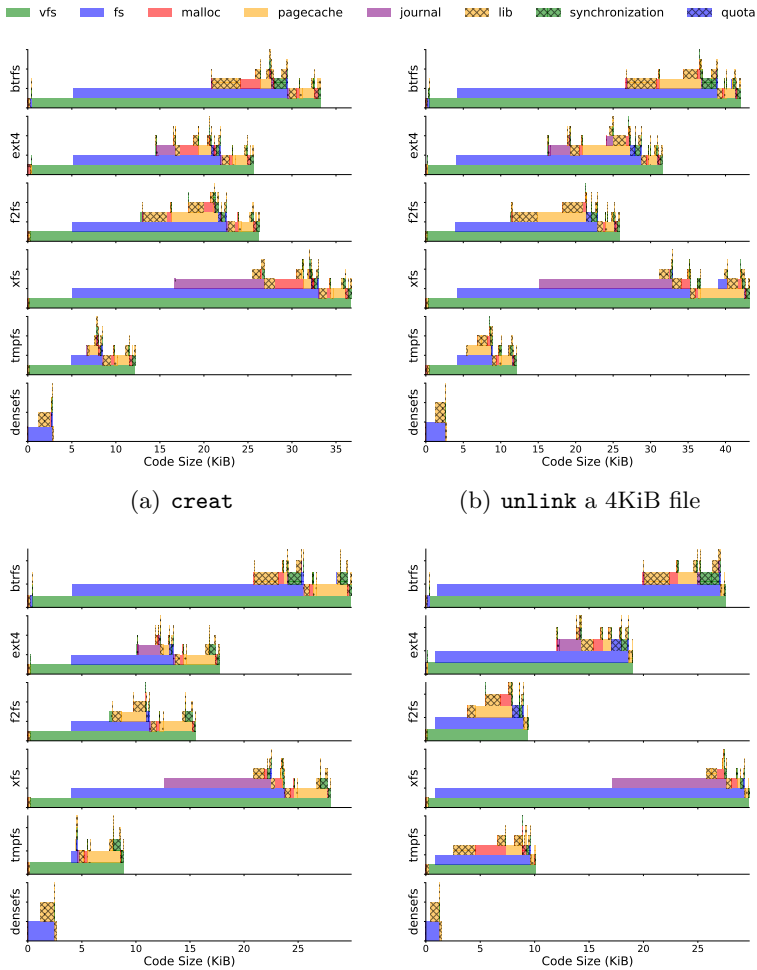
(c) `rename` (d) `fallocate` 1MiB of data

Figure 4.10: Instruction cachemaps of DenseFS, before and after cache-compaction optimizations. The hatched green regions near the tops of the packed cachemaps indicate cache footprint eliminated by the optimizations described in Section 4.3.

fitting all three into two cache lines.

Figure 4.10 shows the effects of the code-compaction techniques described in this section. While the cache-footprint savings provided by these optimizations are small, unlike some of the data-cache optimizations they are essentially "free", with no tradeoffs in the functionality of the filesystem. The code-packing optimizations have the disadvantage, however, of being relatively fragile – small code changes can render a carefully-applied manual optimization moot. A more realistically maintainable option would be to have optimizations like these applied automatically by the compiler, perhaps via a form of profile-guided optimization [112, 115, 158].

Figure 4.11 shows the overall compactness of DenseFS in relation to the same Linux filesystems shown in Figure 4.6. Relative to them, DenseFS reduces code size dramatically, occupying a footprint 3.3-6.4× smaller than even the smallest existing Linux filesystem. In three of the four system calls examined (`creat`, `unlink`, and `rename`), DenseFS's code footprint in its entirety is smaller than the footprint of the VFS code alone in each other filesystem (2.7-3.0KiB, as compared to 3.7-4.8KiB of VFS code). The exception is `fallocate`, which operates on an already-open file descriptor (whereas `creat`, `unlink`, and `rename` operate on files by path), and thus has less work to do in the VFS before being dispatched



(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.11: Cgstack flame graphs showing the code footprint of densefs in comparison to those of Linux filesystems.

into filesystem-specific code. Even in this case, however, at 1.5KiB DenseFS's code footprint is only 16% of that of the next most compact filesystem (f2fs, at 9.4KiB).

4.3.3 A Second Generation

The main practical drawback of DenseFS as presented thus far is its lack of integration with existing filesystem interfaces. This segregation manifests in its implementation, requiring dedicated code to manage its own special file descriptor table, but more problematically in requiring applications to use specialized DenseFS system calls to access it. While in some simple cases this can be made to happen transparently via an external shim layer such as an `LD_PRELOAD` library, this approach rapidly hits its limits when applications do non-trivial or unexpected things with their file descriptors (even something as simple as using `dup2` to duplicate an existing file descriptor at a specific desired number).

Thus, to avoid the necessity of modifying applications or employing such fragile shim layers to support DenseFS, we wish to better integrate it with the kernel's existing filesystem code. We achieve this with a second-generation implementation of DenseFS called DenseFS2. (The initial design is henceforth correspondingly referred to as DenseFS1.) DenseFS2 uses much of the same core code as DenseFS1, but makes

some modifications to existing kernel code in order to allow it to be accessed via normal, non-DenseFS-specific filesystem syscalls and store its file descriptors in each process's existing file descriptor table alongside "normal" ones.

The mechanism we use for this hinges on the simple technique of "borrowing" a bit from a pointer to distinguish between VFS and DenseFS2 objects at runtime. Given the alignment requirements for a particular data structure, it is guaranteed that all pointers to an instance of such a data structure will have some number of bits at their least-significant end that are all zeros. These bits can then be used to encode auxiliary information; when the pointed-to data needs to be referenced, it can be by simply masking off the borrowed low bits and dereferencing the resulting (original) pointer. This technique, often referred to a *tagged pointer*, is commonly used in software such as programming language interpreters [48, 67], and even elsewhere in the Linux VFS, in which a single `unsigned long` is used to store both a pointer to a `struct file` and two metadata flags pertaining to it (DenseFS2 essentially just borrows one additional bit in this same value).

DenseFS2 uses this borrowed pointer bit to add a layer of indirection to some key functions in the Linux VFS. Most prominently, the function `fdget`, used to retrieve a `struct`

`file` pointer corresponding to a given file descriptor number, is altered to instead return a pointer to a new type, `struct qfile`. A `struct qfile` is itself a dummy struct with no members; it simply serves as a unique pointer type to clarify exactly what the semantics of each variable are in the code that handles them. A pointer to a `struct qfile` is in fact a pointer to either a regular Linux `struct file` or a pointer to its DenseFS2 counterpart; the two are distinguished by the borrowed flag in its lower bits. Each point in the VFS code that retrieves a file object from a file descriptor thus checks this bit and either continues on to the regular VFS code or instead dispatches the requested operation to its DenseFS2 equivalent. Because the transition from a `struct qfile` to a `struct file` or an DenseFS2 file is simply a bit manipulation (as opposed to an actual indirection through memory, i.e. a pointer dereference), this layer of indirection imposes little additional overhead.

This mechanism addresses the VFS/DenseFS2 demultiplexing problem for filesystem access via file descriptors, but there is still a corresponding problem for access via path names. DenseFS2 solves this by reserving a special path prefix, `"@@"` (two "at" signs), to indicate that the path name following it should be looked up within DenseFS2. DenseFS2 is thus still not fully integrated into the regular filesystem (it

cannot be mounted as a subtree at an arbitrary location), but instead lives in its own neighboring parallel namespace. This arrangement is somewhat incongruous in the world of Unix-like filesystems, being more analogous to the notion of "drive letters" in the Windows filesystem. Nevertheless, it provides a vastly simpler mechanism for application interoperability than the original DenseFS's set of dedicated system calls. By intercepting execution from existing filesystem syscalls, DenseFS2 allows applications to use it without modification, simply by specifying `"@@"`-prefixed paths to access.

Smaller Inodes

DenseFS2 also includes some further efforts at data cache footprint reduction via inode compaction. While some compromises were made to achieve the 56-byte inode structure used in DenseFS1 (Figure 4.7), there are still some opportunities for additional size reduction.

The `data` union stores a pointer to a red-black tree of data chunks for regular files (8 bytes) or a linked list of entries for directories (two pointers, or 16 bytes). DenseFS2's usage of directory entries does not require the last entry to be immediately accessible, however, so we can instead use an instance of `struct hlist_head`, which contains only a single pointer to the first entry in the list. Because the `dirents`

member was the larger of its two members, this shrinks the `data` union (and thus the entire DenseFS2 inode structure) by 8 bytes.

Another alternative data structure that, like `struct hlist_head`, is already available in the Linux kernel is the bit spinlock – a spinlock implementation that provides mutual exclusion semantically equivalent to a normal spinlock, but uses only a single bit of memory instead of the 32 used for the default spinlock implementation. Though it is documented as being significantly slower than the normal spinlock and may aggravate scalability bottlenecks in situations where there is heavy contention between CPU cores for access to shared inodes, in order to pursue the primary goal of compactness we eliminate the `spinlock_t` and instead borrow the highest bit of the 8-byte `size` field to serve as the new lock for DenseFS2's inode structure, saving another four bytes.

Borrowing a bit from `size` field in this way leaves 63 bits in which to represent a file's size. This representation is still enough to support files up to nearly 8EiB (9,223,372,036,854,775,807 bytes) in size, far larger than seems likely be useful for the foreseeable future. We thus opt to compromise a little further on this parameter and borrow another 16 bits from it,[2] into

[2]The common usage of the term "borrowing" for this practice in programming is curious, implying a promise that the bits will be "returned" at some point in the future. Unfortunately for the rightful owners of such

which we move the `meta_idx` field (the index into the global `<uid, gid, mode>` table). The remaining 47 bits are still sufficient for a healthy 128TiB maximum file size.

Along similar lines, the 8-byte `mtime` and `ctime` fields, consuming an increasing fraction of the remaining size of the DenseFS2 inode as the rest of it shrinks, are next on the list to sacrifice spare bits. At the least-significant end of these fields, the `ktime_t` type's nanosecond resolution is a convenient feature, but is likely more precise than is required for most workloads. At the most-significant end lie a handful of bits that will remain zero for centuries to come – 2^{64} nanoseconds amount to over 584 years. We thus sacrifice some bits from both the low and the high ends of this field, giving up some range and some precision. We discard 21 of the rightmost bits and 3 of the leftmost, leaving a 40-bit timestamp field representing a roughly 73-year range at a resolution of slightly over 2 milliseconds. This new representation is implemented as a struct with two members (one byte and four bytes), to which we apply GCC's `packed` type attribute to ensure it does not contain any padding bytes (which it otherwise would to pad its size out to a multiple of four bytes for alignment purposes).

The final DenseFS2 inode structure is shown in Figure 4.12.

---

bits, this promise is rarely kept.

```
struct densefs2_time {
        uint32_t __low; /* 4 bytes */
        uint8_t __high; /* 1 byte */
} __attribute__((packed));

struct densefs2_inode {
        uint16_t nlink; /* 2 bytes */
        struct densefs2_time mtime; /* 5 bytes */
        struct densefs2_time ctime; /* 5 bytes */
        refcount_t refcount; /* 4 bytes */
        unsigned long __lock_metaidx_size; /* 8 bytes */
        union {
                struct hlist_head {
                        struct hlist_node *first;
                } dirents;
                struct rb_root {
                        struct rb_node *rb_node;
                } chunks;
        } data; /* 8 bytes */
};
```

Figure 4.12: 32-byte DenseFS2 inode structure. The `__lock_metaidx_size` field contains three sub-fields as indicated by its name: a 1-bit spinlock, a 16-bit index into the global `<uid, gid, mode>` table, and a 47-bit size. These are extracted and updated by a set of helper functions that perform the requisite shifting and masking.

With all of these additional inode compactions applied, we reach another key threshold: 32 bytes, allowing a single data cache line to contain two complete DenseFS2 inodes. With this step arises a micro-scale version of the locality problem that many existing disk-oriented filesystems (such as FFS [96]) grapple with. Now that multiple inodes fit in each cache line, careful placement of "related" inodes (those which are likely to exhibit temporal locality in their access patterns) in the same cache lines could potentially yield a benefit for operations that access both. One could, for example, co-locate a directory inode and an inode pointed to by one of its entries in the same cache line, reducing the data cache footprint of path lookups that traverse both. However, with space for only one extra inode to be added alongside another, the ratio of implementation complexity to the expected benefit of doing so seems unappealingly high, and DenseFS2 does not currently make any particular effort to exploit this potential. However, given the nature of its allocation patterns (specifically, that consecutive allocations generally tend to be placed in neighboring locations), access patterns that mirror the order of file and directory creation patterns may tend to serendipitously benefit from this cache locality anyway.

By borrowing some bits from from the pointers in the `data` union and compromising further on range or resolution

in some combination of the link count, reference count, timestamps, maximum file size, and maximum supported number of distinct `<uid, gid, mode>` table entries, the inode structure could potentially be reduced in size even further. 44 bits of the `data` pointer union would probably need to remain (given alignment and the virtual address format, up to 20 could be borrowed on current x86-64 systems), and one bit is required for the spinlock.[3] For a given target inode size, this would then leave a fixed number of bits to allocate at the will of the designer between the remaining fields. In a specialized use-case in which more significant sacrifices could be made in some of these fields,[4] it may be feasible to achieve an inode size that would allow three or, aggressively, possibly even four inodes to fit in a single 64-byte cache line. Increasing the number of inodes per cache line would increase both the ease and the potential benefit of efforts to cluster specific inodes for locality as described above.

[3] Assuming the retention of a fine-grained locking strategy; switching to a single global lock could eliminate this field entirely, but the sacrifice seems likely to be too large for a savings of a single bit.

[4] Such a filesystem could, for example, offer `mkfs` options to allow the administrator to determine the functionality limitations their system can live with.

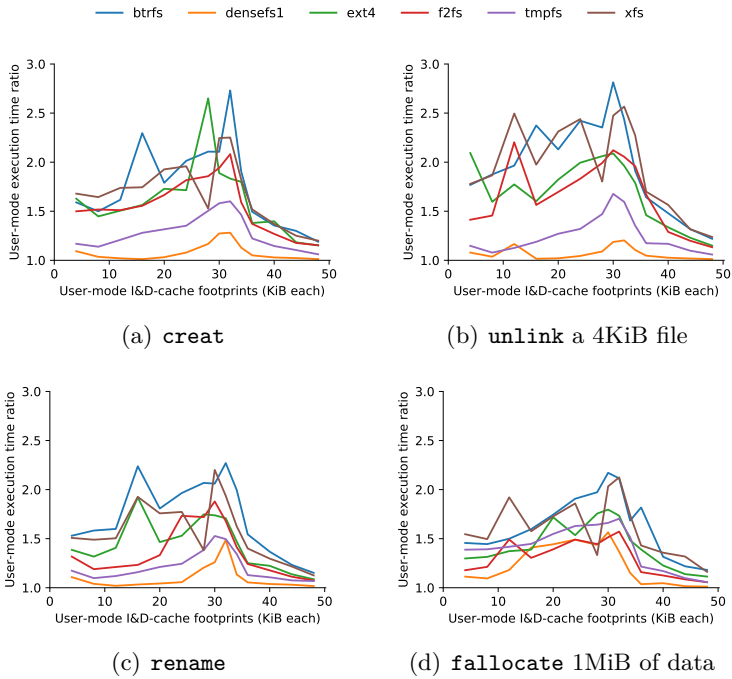
4.4 Evaluation

We evaluate DenseFS1's effectiveness in reducing overall cache pollution using a finely-parameterized synthetic microbenchmark to measure system call impact on user-mode CPU performance. We have also performed experiments running real applications (`grep` and SQLite) on DenseFS1 and DenseFS2; the results of all of these experiments are presented in this section. All measurements were taken with an Intel Xeon E5-2670 CPU running a 4.13-series Linux kernel.

4.4.1 Microbenchmark results

Our microbenchmark tool exercises a single system call at a time, and offers the ability to execute an amount of user-mode "think-time" code in between each instance of the system call. This user-mode code is JIT-compiled before the main loop, and is parameterized to allow adjustment of its instruction and data cache footprints. The microbenchmark reports fine-grained performance statistics for the system call and the user-mode code independently.

Using this tool, we executed system calls and measured the performance of the user code while varying its cache footprint, and comparing the results against the performance of executing the same user code with no system calls at all.



(a) `creat`

(b) `unlink` a 4KiB file

(c) `rename`

(d) `fallocate` 1MiB of data

Figure 4.13: Microbenchmark performance results. The vertical axis shows the relative increase in time spent executing user-mode code when regular calls to the given system call on the given filesystem are inserted (i.e. the performance penalty of the syscall on user-mode execution). The horizontal axis shows the data and instruction cache footprints (both are adjusted in tandem) of the user-mode code executed between system calls.

This comparison allows us to directly measure the system call's impact on the performance of user-mode execution. Figure 4.13 shows the results; a datapoint at 2.0 on the vertical axis means that a user-mode workload with instruction and data working-set sizes indicated by the horizontal position of the datapoint required twice as long to execute when the filesystem operation in question was inserted between iterations.

In almost all cases in these graphs, DenseFS1 incurs the smallest penalty on user-mode performance, in many cases by a wide margin. The `fallocate` graph (Figure 4.13(d)) is the furthest outlier in this regard. While the DenseFS1 line in this graph is still the lowest at most working set sizes, it is generally by a narrower margin, and there are points at which it is not. While this is not ideal, it is consistent with the data from our previous analyses. Figure 4.9(d) shows that for this operation DenseFS1's data cache footprint is significantly larger than for its other operations; this is an artifact of its simple but inefficient bitmap-based space allocation (an aspect of the filesystem's current implementation that is not conducive to its goals). Additionally, f2fs in some cases beating DenseFS1 is consistent the data in Figure 4.6(d), where f2fs showed the smallest code footprint of the existing Linux filesystems – slightly smaller even than tmpfs, which has a relatively

compact code footprint for `fallocate` but a data footprint of 413 cache lines (Figure 4.5(d)) in comparison to f2fs's 127 (Figure 4.3(d)).

The trend across all four operations is for a peak in relative execution-time penalty at a working-set size of 32KiB. This peak makes intuitive sense; at that size, the system-call free user code still fits in the L1 caches, but occupies them entirely. Introducing competition for that cache space in the form of system calls thus pushes the combined workload into experiencing relatively frequent cache misses, where previously there were few to none. Beyond this size the performance penalty of the added filesystem operations tapers off, as the user-mode code already exceeds the capacity of the L1 caches and thus will already be experiencing misses of its own, so the additional ones incurred by the cache perturbation from the system calls are a less dramatic difference.

The expected performance advantage of DenseFS1 over other filesystems thus decreases as application working set size exceeds the capacity of the L1 caches. Even among the rightmost datapoints in Figure 4.13, however, DenseFS1's advantage is still noticeable, incurring execution-time increases of only 1-2% where some other filesystems are still imposing penalties 10-20%.

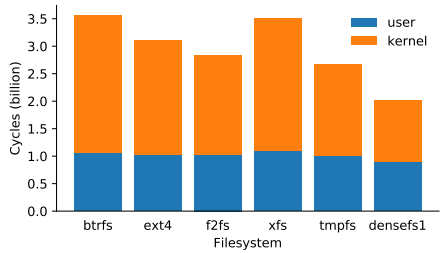


Figure 4.14: User- and kernel-mode CPU cycle counts for `grep -r` on a 750MB directory tree.

4.4.2 DenseFS1 application results: `grep`

To evaluate DenseFS1's performance on a real-world program, we ran version 3.1 of GNU `grep`, using an `LD_PRELOAD` library to redirect its system calls to their DenseFS1 equivalents. We ran `grep` recursively over a directory tree containing 242,272 files and 17,180 directories totaling roughly 750MB of space as measured by `du --apparent-size` (varying slightly between filesystems due to differences in the space consumption of directories); execution-time results are shown in Figure 4.14. Before taking measurements of each filesystem we ran the workload on it once to warm up the page cache so as to make the measured executions operate entirely out of memory and thus be completely CPU bound. Using `perf stat`, we found that DenseFS1 is highly effective at reducing L1 instruction

cache misses. Whereas xfs suffered 84.1M misses on this workload (the most of the five other filesystems tested) and tmpfs 49.0M (the least), DenseFS1 incurred only 1.3M, a reduction of 97% relative to tmpfs (all measurements averaged over five runs each). This improvement allowed `grep`'s user-mode IPC to increase 13% over tmpfs and 18% over xfs.

4.4.3 DenseFS2 application results: SQLite

DenseFS2 being accessible via standard system calls makes it a far simpler target on which to execute an arbitrary application – no source code modifications or `LD_PRELOAD` hacks are needed. Here we use this flexibility to run an unmodified benchmark using SQLite, a popular embedded SQL database employed in a wide variety of systems, including major mobile operating systems, web browsers, and embedded systems [60]. The benchmark is a simple workload, based on one previously used in a study of SSD performance [59], that transactionally inserts random keys into a two-column key-value table.

As DenseFS2's feature set is not entirely at parity with those of existing filesystems, our benchmark issues a pair of `PRAGMA` statements to configure SQLite so as to level the playing field. First, since we aim to evaluate performance operating on memory-like storage and hence the `fsync` family of operations are no-ops on DenseFS2, we disable all such dura-

bility system calls so that all file accesses remain in-memory operations. Second, we override the default SQLite "vfs"[5] setting. By default SQLite uses file locking operations for concurrency control. DenseFS2, however, does not implement any form of file locking, so we instead configure SQLite to use one of two alternate vfs settings that do not require it. The `unix-dotfile` vfs uses a dedicated lock file (actually a lock directory) instead of explicit locking operations; the `unix-none` vfs simply omits all locking operations, relying on the assumption that no other processes will be concurrently accessing the database. We additionally experiment with four different settings of SQLite's `journal_mode` parameter (`off`, `truncate`, `delete` and `persist`), which cause it to use different filesystem operations in its commit protocol [61].

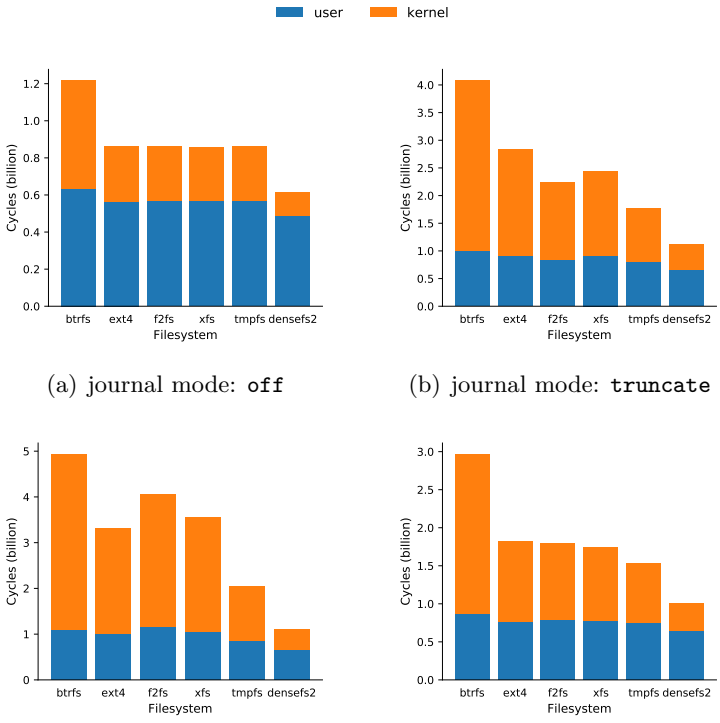
In order to study the effects on different patterns of user-mode execution, we have implemented the same benchmark in both C and Python (the latter executed using version 3.6.5 of the CPython interpreter, both using version 3.21.0 of SQLite). Python's `sqlite3` module is written in C and hence calls the native SQLite library code fairly directly for the bulk of its work. The bytecode interpretation of the benchmark's Python code, however, is still a sufficient fraction of overall execution

[5]Note that while it is conceptually similar, the "vfs" referred to here is a configurable abstraction internal to SQLite itself and completely independent of the kernel VFS layer discussed elsewhere in this chapter.

to lead to a nontrivial difference in the executed user-mode code; the Python version of the benchmark program executes 41-48% more user instructions than the C version to perform the same number of operations.

Figures 4.15 through 4.18 show performance results of both versions of our benchmark performing 16,384 insert operations on the five Linux filesystems we have studied and DenseFS2. As expected, comparing the C and Python versions of the benchmark, we see that in similar configurations they consume similar numbers of kernel-mode CPU cycles, differing primarily in their usage of user-mode CPU time. In all cases, however, DenseFS2 achieves overall performance significantly higher than any other filesystem we have evaluated.

In the closest case (the smallest speedup of DenseFS2 over any other filesystem), DenseFS2 reduces the Python benchmark's execution time by nearly 20% in comparison to f2fs with journaling disabled (the `off` mode) and the `unix-none` vfs (Figure 4.16(a)). This case demonstrating the smallest performance gain makes sense; with no lock-directory creation and deletion, no filesystem activity for journaling operations, and the added user-mode overhead of the Python interpreter, it is proportionally the least filesystem-intensive of the configurations we measured. Despite this, DenseFS2 nevertheless achieves its overall overall speedup not only by reducing



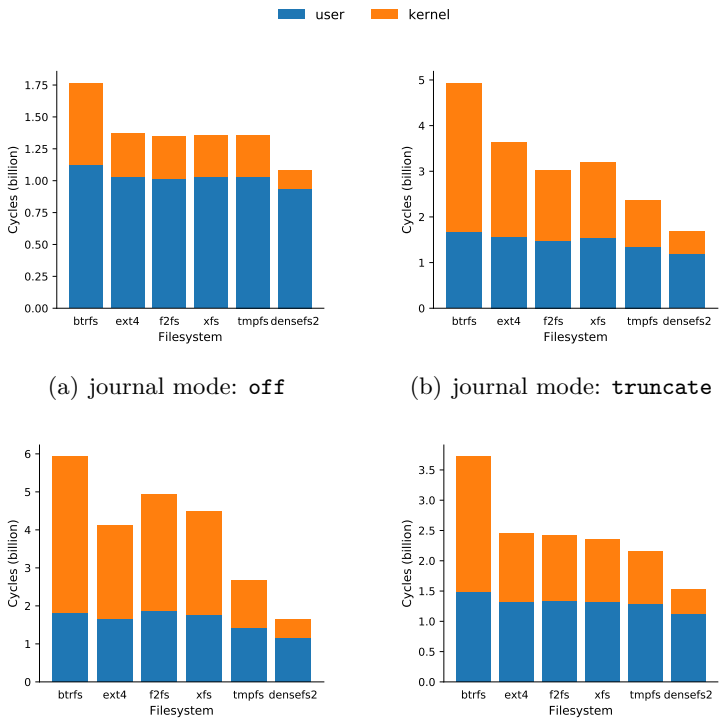
(a) journal mode: `off`

(b) journal mode: `truncate`

(c) journal mode: `delete`

(d) journal mode: `persist`

Figure 4.15: User- and kernel-mode CPU cycle counts for SQLite random-insert benchmark with the `unix-none` vfs, C version.



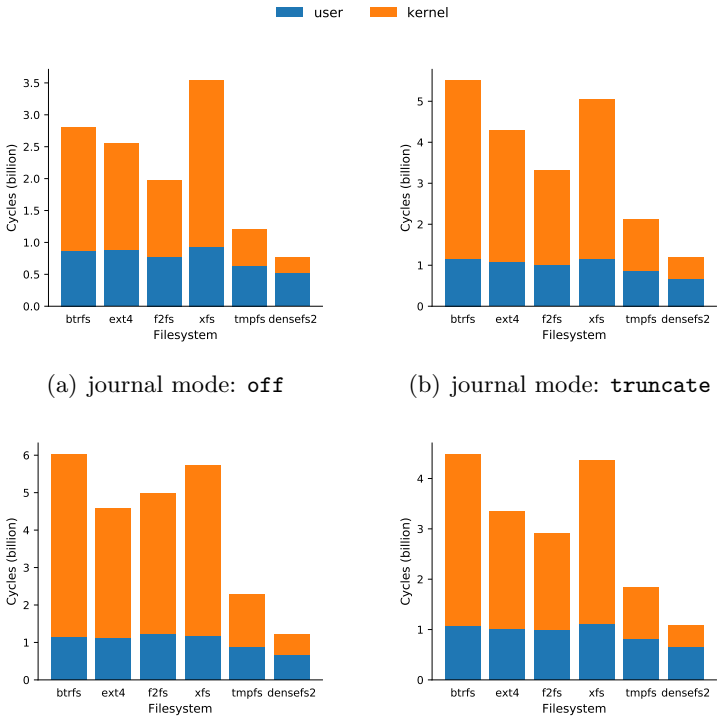
(a) journal mode: `off`

(b) journal mode: `truncate`

(c) journal mode: `delete`

(d) journal mode: `persist`

Figure 4.16: User- and kernel-mode CPU cycle counts for SQLite random-insert benchmark with the `unix-none` vfs, Python version.



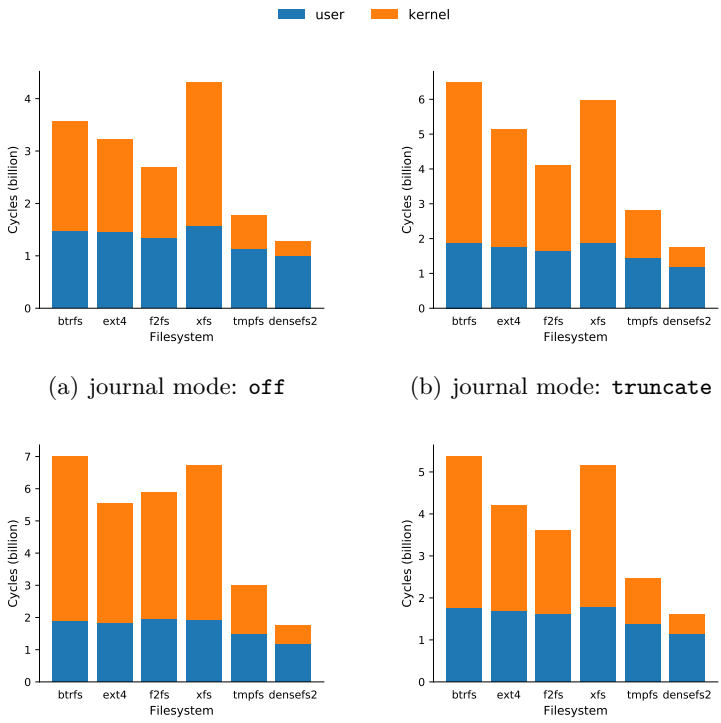
(a) journal mode: `off`

(b) journal mode: `truncate`

(c) journal mode: `delete`

(d) journal mode: `persist`

Figure 4.17: User- and kernel-mode CPU cycle counts for SQLite random-insert benchmark with the `unix-dotfile` vfs, C version.



(a) journal mode: `off`

(b) journal mode: `truncate`

(c) journal mode: `delete`

(d) journal mode: `persist`

Figure 4.18: User- and kernel-mode CPU cycle counts for SQLite random-insert benchmark with the `unix-dotfile` vfs, Python version.

kernel-mode execution time, but also by allowing increased user-mode performance. The benchmark executes the same user-mode code on all filesystems, but achieves 8.8% higher IPC with DenseFS2 than with f2fs.

A much more dramatic effect can be seen at the opposite end of the spectrum, with the C version of the benchmark running with the `delete` journal mode and the `unix-dotfile` vfs (Figure 4.17(c)). Here we see the benchmark executing nearly five times faster on DenseFS2 than it does on btrfs and xfs, and still nearly twice as fast as on tmpfs (the fastest of the existing filesystems). Further, in comparison to f2fs, the benchmark's user-mode IPC increases by 81.7% when run on DenseFS2 (from 0.43 to 0.78). tmpfs fares better in user IPC at 0.6, though it is still well short of DenseFS2's user-mode performance.

4.5 Related Work

DenseFS is not the first filesystem to optimize for the compactness of its data structures. Two examples of existing Linux filesystems which also do this are cramfs [136] and squashfs [86, 87], both of which aim to provide a read-only filesystem using as little storage as possible. These filesystems are designed primarily for use in small embedded systems

where being able to use a smaller EEPROM or flash chip can provide a significant reduction in per-unit costs. They thus differ from DenseFS in that their space optimizations are applied to the data structures that represent the filesystem as stored in the underlying persistent media rather than their in-memory counterparts – by nature of being implemented through the main Linux VFS layer, they use the same in-memory inode structures, directory entries (dentries), and so forth as any other Linux filesystem. Additionally, most of their space optimization is oriented toward reducing the space consumption of file data by compressing it (as opposed to DenseFS's focus on metadata). Nevertheless, some of the compaction techniques used in DenseFS are similar to techniques used in cramfs and squashfs.

In order to reduce the space consumption of its inodes, cramfs omits some fields, sacrificing not just access timestamps as DenseFS does, but all timestamps entirely, as well as link counts (though multiple hard links to the same inode can still be safely allowed due to the read-only nature of the filesystem). Additionally, somewhat analogously to DenseFS's sacrifice of some resolution and range in using five-byte timestamps, cramfs shrinks the inode uid and gid fields from their full 32-bit form to 16 and 8 bits respectively, simply truncating any bits beyond that. It also stores file size in a 24-bit field, imposing

a 16MiB limit on maximum file size.[6] In combination, these design choices allow cramfs to fit its inodes in 12 bytes.

Squashfs was developed somewhat later than cramfs and employs a more sophisticated design that aims to address some of cramfs's shortcomings. It provides a single 32-bit timestamp per inode (which, due to it also being read-only, acts as both the `ctime` and `mtime` fields), a 32-bit link-count field, and increases maximum file size to approximately 2TiB. It also employs a scheme similar to the global `<uid, gid, mode>` table used in DenseFS, but uses it only for uids and gids, and keeps the two separate, storing a distinct index for each. While this does not directly exploit the strong correlation between uids and gids as DenseFS's unified table does, squashfs's inode tables are compressed in bulk using a general-purpose compression algorithm (whereas cramfs compresses only file data), reducing the information redundancy in the final form of the inodes. Due to the compression, squashfs's inodes are not a single fixed size, but consume only 8 bytes on average.

Prior research has proposed techniques for improved cache locality by automated means such as compiler optimizations [26, 29, 66, 112, 115, 151]. While these approaches should be able to achieve similar benefits to what we have done to compact

[6] While this would be problematic for a general-purpose filesystem, it is not necessarily unreasonable for the small embedded systems for which cramfs is designed.

DenseFS's code footprint (and the techniques described in Section 4.3.2 are indeed the same ones these tools automate), DenseFS's data cache optimizations are more aggressive than what can be feasibly performed by an automated tool, because they are deeply dependent on the specific semantics of the operation of a filesystem, and sometimes involve small compromises in functionality. Existing work has also investigated cache-conscious storage systems [55, 89] in a standalone context. However, the nature of a local filesystem sharing a cache with application code presents an interesting and different context, in which optimization for absolute compactness is of greater importance (so as to reduce pollution incurred on arbitrary application code sharing the same cache). DenseFS's global `<uid, gid, mode>` table closely resembles a structure used in Microsoft's NTFS filesystem to reduce the on-disk space consumption of its security descriptors [123]; our technique here is effectively the CPU-cache analog of this sort of compression.

4.6 Conclusion

We have performed a detailed analysis of the memory access patterns of existing Linux filesystems, and found that their cache footprints are generally large enough to cause significant

disturbance to application L1 cache state. We have then shown with DenseFS (versions 1 and 2) that it is possible to implement a filesystem with a much smaller cache footprint than found in existing filesystems. Further, we have seen demonstrated with an array of performance measurements that our cache-compact filesystem can improve performance not only by performing filesystem operations faster, but also by reducing the cache pollution it incurs. We have shown that this has a significant positive effect on the performance of user-mode application code, producing IPC improvements of 8.1-81.7% in the execution of the same user code.

Our initial implementation of DenseFS (DenseFS1) made some trade-offs in functionality in an effort to reduce its code footprint as much as possible. The revised design of DenseFS2, however, attains most of the same benefit while remaining compatible with existing software, providing the same system call interface as existing filesystems while offering sizable performance improvements over them.

Both versions of DenseFS, however, are research prototypes that are far from being able to take the place of an existing filesystem in any real-world usage. Addressing DenseFS's shortcomings in the areas of scalability and feature support while retaining as much of its compactness as possible could be a promising direction for future research.

5

Conclusions

Many significant developments in storage software over the years have been driven by changes in the landscape of contemporary hardware. In the late 1980s, while inexpensive hard disks were readily available, they were becoming increasingly unable to keep up with the much more rapidly improving performance of CPUs and memory. This disparity led to the development of RAID [110], which utilized multiple disks in tandem to achieve large improvements in I/O performance and reliability. A few years later, the widening gap between random and sequential disk I/O performance and growing RAM capacities enabling more caching of disk contents gave rise to LFS [121], which redesigned traditional filesystem data structures to produce more sequential I/O patterns that utilized a

greater fraction of the available disk bandwidth, leaning on large in-memory caches to maintain good read performance. Later, as technology improvements brought disk sizes into ranges where multi-terabyte arrays were commonplace, capacities grew large enough that the frequency of "random" data corruption (due to hardware bit-errors or bugs in disk firmware and device drivers) could become problematic. Concerns about such corruption brought about the development of filesystems like ZFS [22] and btrfs [93, 119] that employ full checksumming of all data and metadata to protect against such faults, and integrated RAID to reduce rebuild times after a drive failure. More recently, the integration of cheap commodity flash storage into billions of consumers' mobile devices has driven the need for filesystems like F2FS [79] that are tailored for this class of hardware. As the state of the art in computing hardware has continued to evolve, the work presented in this dissertation continues in this vein.

5.1 Increasing Core Counts and Trace Replay

The effects of increasing CPU core counts in relation to filesystems have been studied in previous work [24, 34, 100]. In Chapter 2, we explored a second-order effect of this trend

on storage systems via changes in application software. The availability of larger numbers of processors has led to applications tending to employ correspondingly larger numbers of threads, making their I/O patterns not only much more complex, but also highly nondeterministic, which presents a difficult problem for trace replay. Trace replay is a popular technique for evaluating the performance of storage systems, but its utility hinges critically on being able to accurately mimic the behavior of real applications; with complex, nondeterministic multithreaded applications, simplistic replay strategies cannot achieve this.

The ROOT approach and our prototype implementation ARTC address this problem with a novel technique using semantic analysis of the resources referred to in a trace to construct a dependency graph, and use this graph to allow replay to safely diverge from the ordering of events recorded in the trace. This flexibility preserves the nondeterminism of the original application, and our experiments have demonstrated that this allows it to provide a higher-fidelity reproduction of actual application behavior than other replay techniques can achieve.

5.2 Flash and Storage Virtualization

In Chapter 3, we demonstrated with ANViL a storage virtualization system designed for the high-performance flash devices that have seen widespread adoption in recent years. ANViL's design aims to produce I/O patterns that mesh well with the characteristics of the flash storage beneath it; by exposing the address-remapping it uses in doing so, it provides a simple but rich extended block storage interface to the applications and filesystems above it. We focus particularly on the techniques employed in ANViL's garbage collector to deal with the challenges of its many-to-one block address map, scale and performance requirements, and concurrency with foreground I/O activity.

We have shown how the range operations ANViL provides as extensions to the block interface are powerful primitives that can be easily used to implement a number of useful features such as snapshots, file cloning, and efficient transactional updates.

5.3 NVM and Filesystem Cache Behavior

Chapter 4 looks forward to the near future of storage hardware, in which emerging nonvolatile memory technologies

appear poised to become increasingly common. These devices offer dramatic reductions in access latencies, upending the performance assumptions underlying many components of contemporary storage software stacks, and filesystems in particular. We present as prototype filesystems redesigned in light of this change two forms of DenseFS, a more agressive initial version with its own bespoke system calls that allow it to remain completely disentangled from existing filesystem code, and a slightly more pragmatic, compatible design that hooks into existing system calls while reusing as little of their code as possible. While achieving high performance in executing filesystem operations is important, these filesystems additionally focus on another facet of overall system performance in aiming to minimize their impact on the execution performance of application code. They do so by keeping their footprints in the CPU cache as compact as possible, reducing the cache pollution incurred by performing filesystem operations.

Our experiments have shown that DenseFS achieves excellent performance, not only in the speed of its own operations, but also in improving the overall CPU performance seen by application code. By virtue of it being far less destructive toward application cache state, applications running on DenseFS often see their user-mode IPC improve by large ratios. These results have demonstrated the importance of filesystem cache

behavior to overall system performance. While our current DenseFS implementation is not itself a viable filesystem, we hope that the performance phenomena it demonstrates will be considered in the design and implementation of future filesystems developed for low-latency storage.

5.4 Future Work

While the work presented in this dissertation has been sufficient to evaluate the ideas and systems described therein and demonstrate their effectiveness, as is so often the case with research, answering some questions in turn raises additional ones.

ROOT and ARTC demonstrate the potential of non-order-preserving trace replay, but there remain details of its operation that would be worthy of further research. For example, while we have demonstrated that the overconstraint caused by simpler (strictly-ordered) replay methods leads to performance inaccuracy, we have not attempted to analyze the degree to which ARTC's more flexible replay might underconstrain (or potentially still even overconstrain) replay relative to the actual application-level semantics of the original program. And whatever the amount, could it be reduced and more accurate replay be achieved with traces that included additional infor-

mation, such as records of thread synchronization operations? ROOT and ARTC are a step forward in multithreaded trace replay, but there is much road yet to be traversed.

ANViL provides a powerful flash-oriented storage virtualization platform, though its address-remapping structure, with data blocks shared by multiple logical address, leaves a number of unsolved problems. How might secure deletion be implemented? How should space accounting be handled with multiple users of the same physical space? How could an application such as a deduplicator determine whether two identical-looking blocks are in fact already sharing the same physical space? While ANViL is useful in its current form, questions such as these would likely need to be answered before a design of its nature could be reasonably put into real-world use.

DenseFS raises a number of interesting follow-on questions. While it has clearly shown the importance of filesystem cache behavior in overall system performance, a more detailed examination of exactly which factors have what effects could be enlightening (for example, the various different compaction techniques, or the relative effectiveness of instruction and data cache footprint reduction). Additionally, a variety of practical problems lie between the current state of DenseFS and a real-world filesystem, such as the question of how to extend

DenseFS to achieve better scalability and provide features like crash consistency, ideally while retaining as much compactness as possible. Could techniques like profile-guided optimization be used to automate some compaction optimizations so as to achieve a less delicately-arranged, more maintainable codebase? If questions like these can be addressed, a DenseFS-like filesystem might one day be useful in improving application and overall system performance with real NVM hardware.

5.5 Final Thoughts

While the three pieces presented in Chapters 2 through 4 all study evolutions in software brought about by changes in hardware, the relative chronology of the specific hardware changes to which they each relate provides an interesting perspective.

Multicore CPUs have been commonplace for over a decade, and hence the effects we studied in Chapter 2 are of a somewhat delayed, downstream nature. The waves caused by this particular hardware change have had to time to ripple outward, and here we study an echo of them. Application software itself adapted to make use of multicore CPUs via more aggressive use of multithreading; with ROOT and ARTC we in turn address the effects of that change and the new challenges they

present for the tools we use to evaluate storage systems.

Flash storage is now a commodity item, but it has not been established as such for as long as have multicore CPUs. Thus in Chapter 3 the problem we address is more a part of the general retooling of software components across the storage stack – a process that has been well underway for years, but is still decidedly ongoing.

Nonvolatile memory, however, is just beginning to arrive. There are not yet any well-established answers to the question of the "right" way to integrate and manage NVM in the storage stack. Research in this area thus tends to be of a highly speculative, experimental nature; Chapter 4 contains our own contribution to just this sort of experimentation.

These three points form a line that is nicely illustrative of a general pattern in the chronology of hardware-driven software evolution. When a significant shift in hardware is just on the horizon, green-field research around it busily searches for novel ways for software to accommodate and exploit it. After the initial splash of its arrival, its waves propagate outward as adoption becomes widespread and surrounding areas of software gradually adapt to it. Finally, once it is well-established and truly ubiquitous, smaller waves resulting from it, perhaps reflected off of other software components, become interesting research problems in their own right. This pattern

seems likely to continue well into the future as hardware technology improves, software evolves to adapt to it, and the cycle of renewal continues.

Bibliography

[1] Native Flash Support for Applications. http://www.flashmemorysummit.com/.

[2] ioCache. http://www.fusionio.com/products/iocache, 2012.

[3] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[4] Nitin Agarwal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy.

Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[5] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009. USENIX Association.

[6] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003. ACM.

[7] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] Dave Anderson. You Don't Know Jack About Disks. *ACM Queue*, 1(4):20–30, June 2003.

[9] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004. USENIX Association.

[10] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[11] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Ottawa Linux Symposium*, 2009.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research:

A View from Berkeley. Technical report, University of California, Berkeley, 2006.

[14] Jens Axboe. fio: Flexible I/O Tester. `http://git.kernel.dk/cgit/fio/`.

[15] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, 2009.

[16] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991. ACM.

[17] Seth Benton. LevelDB in Riak 1.2. `http://basho.com/posts/technical/leveldb-in-riak-1-2/`.

[18] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages*

and Operating Systems, ASPLOS XV, pages 53–64, New York, NY, USA, 2010. ACM.

[19] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[20] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[21] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. ACM.

[22] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, 2007.

[23] Dhruba Borthakur. RocksDB: A persistent key-value store. http://rocksdb.org, 2014.

[24] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[25] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[26] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious Data Placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 139–149, 1998.

[27] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In

ASPLOS XVII: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2012. ACM. 415125.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[29] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, 1999.

[30] Dave Chinner. XFS Delayed Logging Design. `https://www.kernel.org/doc/Documentation/filesystems/xfs-delayed-logging-design.txt`.

[31] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)*, 4(4), February 2009.

[32] Sung-Eun Choi and E. Christopher Lewis. A Study of Common Pitfalls in Simple Multi-threaded Programs. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 325–329, New York, NY, USA, 2000. ACM.

[33] A.N.M. Imroz Choudhury. *Visualizing Program Memory Behavior Using Memory Reference Traces.* PhD thesis, University of Utah, 2012.

[34] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, 2013.

[35] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, pages 105–118, Newport Beach, California, March 2011.

[36] Kees Cook. Kernel Address Space Layout Randomization. Linux Security Summit, 2013.

[37] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[38] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[39] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, 2014.

[40] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: Flexible, Efficient File Volume

Virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[41] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 203–216, Berkeley, CA, USA, 2003. USENIX Association.

[42] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the 17th Annual Large Installation System Administration Conference (LISA '03)*, San Diego, California, October 2003. USENIX Association.

[43] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[44] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study

of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, 2012.

[45] Annie Foong and Frank Hady. Storage as fast as rest of the System. In *2016 IEEE 8th International Memory Workshop*, Paris, France, May 2016.

[46] Gregory R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[47] Gregory R. Ganger and Yale N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Transactions on Computers*, June 1998.

[48] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[49] R.P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[50] Jim Gray and Bob Fitzgerald. Flash Disk Opportunity for Server Applications. *ACM Queue*, 6(4):18–23, July 2008.

[51] Brendan Gregg. The Flame Graph. *ACM Queue*, 14(2):10:91–10:110, March 2016.

[52] Fanglu Guo and Petros Efstathopoulos. Building a High-performance Deduplication System. In *USENIX Annual Technical Conference*, 2011.

[53] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[54] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[55] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-conscious Storage Technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 417–428, 2003.

[56] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM.

[57] Red Hat. Device-mapper Resource Page. `https://sourceware.org/dm/`.

[58] Red Hat. LVM2 Resource Page. `http://www.sourceware.org/lvm2/`.

[59] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *EuroSys '17*, Belgrade, Serbia, April 2017.

[60] D. Richard Hipp. Most Widely Deployed SQL Database Engine. `https://www.sqlite.org/mostdeployed.html`.

[61] D. Richard Hipp. Pragma statements supported by SQLite. `https://www.sqlite.org/pragma.html`.

[62] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In

Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94), San Francisco, California, January 1994.

[63] Micha Hofri. Disk Scheduling: FCFS vs. SSTF Revisited. *Communications of the ACM*, 23(11):645–653, November 1980.

[64] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, February 1988.

[65] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional Storage Virtualization. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 14–24, 2004.

[66] Wen-mei W. Hwu and Pohua P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 242–251, 1989.

[67] Roberto Ierusalimschy and Luiz Henrique De Figueiredo. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 2005.

[68] Inktank Storage, Inc. KeyValueStore Config Reference. `http://docs.ceph.com/docs/hammer/rados/configuration/keyvaluestore-config-ref/`.

[69] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[70] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. Enabling TRIM Support in SSD RAIDs. Technical report, Department of Computer Science, University of Rostock, 2011.

[71] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[72] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and Efficient Replaying of File System Traces.

In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005. USENIX Association.

[73] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Memory Management*, pages 103–115. Springer, 1992.

[74] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, 2018.

[75] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, Santa Clara, CA, February 2015. USENIX Association.

[76] Michael Kluge, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel. Pattern Matching and I/O Replay for POSIX I/O in Parallel Programs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09. Springer-Verlag, 2009.

[77] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS '10*, New York, NY, June 2010. ACM.

[78] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *Computer*, 27(10):15–26, October 1994.

[79] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, 2015.

[80] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[81] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, 2009.

[82] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008. USENIX Association.

[83] Adam H. Leventhal. A File System All Its Own. *Communications of the ACM*, 56(5):64–67, May 2013.

[84] Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. CloudProphet: Towards Application Performance Prediction in Cloud. In *SIGCOMM '11*, Toronto, Canada, August 2011. ACM.

[85] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, 2011.

[86] Phillip Lougher. An Overview of the SquashFS filesystem. `https://elinux.org/images/3/32/Squashfs-elce.pdf`.

[87] Phillip Lougher. SQUASHFS 4.0 FILESYSTEM.

`https://www.kernel.org/doc/Documentation/filesystems/squashfs.txt`.

[88] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 75–88, Santa Clara, CA, 2014. USENIX.

[89] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[90] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, Santa Clara, CA, July 2015. USENIX Association.

[91] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *6th USENIX Workshop on Hot Topics in Storage and*

File Systems (HotStorage 14), Philadelphia, PA, June 2014. USENIX Association.

[92] K. Maruyama and S. E. Smith. Optimal Reorganization of Distributed Space Disk Files. *Communications of the ACM*, 19(11):634–642, November 1976.

[93] Chris Mason. Btrfs Design. `http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html`, 2011.

[94] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[95] J. May. Pianola: A Script-based I/O Benchmark. In *Petascale Data Storage Workshop*, November 2008.

[96] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[97] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings*

of the USENIX Annual Technical Conference (USENIX '99), Monterey, California, June 1999.

[98] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '91)*, pages 33–43, Dallas, Texas, January 1991.

[99] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007. USENIX Association.

[100] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[101] Andrew Morton. Re: [PATCH v10 00/21] Support ext4 on NV-DIMMs. https://lwn.net/Articles/610182/.

[102] Mark Moshayedi and Patrick Wilkison. Enterprise SSDs. *ACM Queue*, 6(4):32–39, July 2008.

[103] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[104] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. ptrim() + exists(): Exposing New FTL Primitives to Applications. In *Proceedings of the Non-Volatile Memory Workshop*, NVMW '11, 2011.

[105] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *Journal of Parallel and Distributed Computing*, August 2009.

[106] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):26–29, September 2005.

[107] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on*

Operating System Principles (SOSP '85), Orcas Island, Washington, December 1985. ACM.

[108] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *HPCA*, pages 301–311. IEEE Computer Society, 2011.

[109] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. GIGA+: Scalable Directories for Shared File Systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[110] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, 1988.

[111] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[112] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, 1990.

[113] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[114] POSIX.1-2008. The Open Group Base Specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[115] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code Layout Optimizations for Transaction Processing Workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 155–164, 2001.

[116] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual

Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 31–39, Palo Alto, California, 1991.

[117] Erik Riedel. Storage Systems: Not Just a Bunch of Disks Anymore. *ACM Queue*, 1(4):32–41, June 2003.

[118] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7), July 1974.

[119] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Transactions on Storage*, 9(3):9:1–9:32, August 2013.

[120] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000. USENIX Association.

[121] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[122] Kostadis Roussos. Storage Virtualization Gets Smart. *ACM Queue*, 5(6):38–44, September 2007.

[123] Mark Russinovich. Inside Win2K NTFS, Part 1. `https://msdn.microsoft.com/en-us/library/ms995846.aspx`.

[124] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[125] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, 2012.

[126] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010. USENIX Association.

[127] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX*

Winter Technical Conference (USENIX Winter '90), pages 313–323, Washington, D.C, January 1990.

[128] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage Virtualization: Integration and Load Balancing in Data Centers. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, 2008.

[129] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, Seattle, Washington, June 1997. ACM.

[130] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The Missing Memristor Found. *Nature*, 453:80–83, 2008.

[131] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[132] Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, and Sang Lyul Min. LTFTL: Lightweight Time-shift Flash Translation Layer for Flash Memory Based Embedded Storage. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, 2008.

[133] Herb Sutter and James Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, September 2005.

[134] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[135] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting Flexible, Replayable Models from Large Block Traces. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

[136] Linus Torvalds and Nicolas Pitre. Cramfs - cram a filesystem onto a small ROM. `https://www.kernel.org/doc/Documentation/filesystems/cramfs.txt`.

[137] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem. In *The Fourth Annual Linux Expo*, Durham, NC, USA, May 1998.

[138] E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston. A Cache Visualization Tool. *Computer*, 30(7):71–78, Jul 1997.

[139] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, California, February 2011.

[140] Bill Venners. *Inside the Java virtual machine*. McGraw-Hill, Inc., 1996.

[141] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. `http://www.veritas.com/us/products/volumemanager/whitepaper-02.html`, July 2005.

[142] Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of*

the Ninth European Conference on Computer Systems, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[143] Haris Volos and Michael Swift. Storage Systems for Storage-Class Memory. In *Proc. of Annual Non-Volatile Memories Workshop (NVMW'11)*, 2011.

[144] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, Newport Beach, California, March 2011.

[145] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[146] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012. USENIX Association.

[147] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, 1992. Springer-Verlag.

[148] Darrick J. Wong. Ext4 Disk Layout. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`.

[149] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 241–251, 1994.

[150] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[151] Youfeng Wu. Ordering Functions for Improving Memory Reference Locality in a Shared Memory Multiprocessor System. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 218–221, 1992.

[152] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[153] Neeraja J. Yadwadkar, Chiranjib Bhattacharyya, K. Gopinath, Thirumale Niranjan, and Sai Susarla. Discovery of Application Workloads from Network File Traces. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010. USENIX Association.

[154] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, page 10. ACM, 2013.

[155] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, Oct 2014. USENIX Association.

[156] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, 2013.

[157] Y. Yu, K. Beyls, and E. H. D'Hollander. Visualizing the Impact of the Cache on Program Execution. In *Proceedings Fifth International Conference on Information Visualisation*, pages 336–341, 2001.

[158] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in Profile-guided Operating System Kernel Optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 4:1–4:6, 2014.

[159] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, 2016. USENIX Association.

[160] Da Zheng, Randal Burns, and Alexander S. Szalay. A Parallel Page Cache: IOPS and Caching for Multicore Systems. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, 2012. USENIX.

[161] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.