Emergent Properties in Modular Storage: a Study of Apple Desktop Applications, Facebook Messages, and Docker Containers

By

Tyler Harter

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: August 24th, 2016

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, Associate Professor, Computer Sciences
Jignesh M. Patel, Professor, Computer Sciences
Colin N. Dewey, Associate Professor, Biostatistics

To my parents

Acknowledgments

In this section, I want to thank the many people who have supported me in my graduate work culminating in this PhD, and in my life more generally. Family, friends, and faculty have helped me succeed and (more importantly) grow as person.

Andrea and Remzi: A student could not ask for better advisors than Andrea and Remzi. Many view earning a PhD as a reward for suffering through graduate school, but in my case, it feels like the reward was the opportunity to work for Andrea and Remzi, and earning a PhD at the end is merely a bonus. Regular meetings with my advisors were full of energy and creativity. These meetings were often the highlight of my week.

This dissertation is a study of emergent properties, or characteristics that show up in a combination that could not be predicted by the subcomponents. Many emergent properties are undesirable, but the collaboration between Andrea and Remzi is a rare example of when "a whole is much greater than a sum of the parts." A student would be lucky to work with either of them individually, but the dynamic that occurs when they co-advise students and co-author papers is truly phenomenal. Their differing perspectives has given me a much richer understanding of systems, and their lighthearted banter makes meetings fun and entertaining. Andrea and Remzi have very different styles, and I have learned a great deal from each of them.

My first introduction to systems was through Remzi when I took his operating systems class as an undergraduate at UW-Madison. Remzi has a gift for teaching. He takes complex and seemingly dry material and transforms it into something that is understandable and interesting, and his killer sense of humor has turned class into a new form of entertainment. Many people choose their field of study, then find a mentor in that field. In my case, I wanted to work with Remzi, and so I became a systems researcher. Remzi is a visionary professor, and under his mentorship I feel my own creativity has flourished. Remzi has taught me that data is beautiful and shown me that if one can find the right way to visualize a pattern, most problems suddenly become very clear. Remzi also showed me how to work with industry in order to learn about "real-world" problems and choose impactful research projects. When I have had the opportunity to teach myself, Remzi provided much indispensable advice on how to connect with students and present well.

Andrea's mentorship has taught me how to write well and organize my thoughts. During my first research project, we went through well over a dozen iterations of feedback from Andrea on the writing. Each iteration, she would write many pages of feedback, suggesting not only specific changes, but explaining the thought process behind those changes. At the time, I was puzzled by her approach: it would clearly have taken her much less time to simply edit herself rather than explain everything in such detail. Now, I understand that Andrea is truly a student-first professor: her goal was not to write the best possible paper, but to turn me into the best possible writer. Andrea is incredibly practical and empathetic. She has an uncanny sense of not only the benefits of taking a research project in a given direction, but also of the student effort that would be required to take that direction. Her intuition helps students maximize the impact-to-effort ratio. She has enabled me to be very productive in grad school while still having time for a life outside of work.

Faculty: Many other professors and faculty have been a very important part of my time spent at UW-Madison as a student. I would like to thank Mike Swift, Jignesh Patel, and Colin Dewey for being on my committee and for their excellent teaching of the Distributed Systems, Advanced Databases, and Advanced for Bioinformatics courses, respectively. Mike in particular has provided very useful feedback on several of my research projects. Also, when I was preparing for my qualifying exam, he was extremely generous with his time, answering questions about the papers and discussing possible answers to old questions. I also thank Aditya Akella, Suman Banerjee, AnHai Doan, Ben Liblit, and Jim Skrentny for their excellent courses and teaching me so much.

Industry Collaborators: Much of the work in this dissertation was done in collaboration with industry. Most of the work in Chapter 3 was done during two internships at Facebook, where many people were incredibly helpful. First, I thank Dhruba Borthakur for being an excellent manager and teaching me so much. Dhruba is an incredibly good hacker, yet he is very in touch with work happening in the academic community. He is very thoughtful and projects a contagious enthusiasm. I also thank Liyin Tang for helping us get tracing access, Amitanand Aiyer for his help interpreting results, and Siying Dong and Nagavamsi Ponnekanti for their very helpful code reviews. I also thank Harry Li for coordinating many details of the collaboration. Finally, I thank Venkateshwaran Venkataramani for his invaluable advice on the OpenLambda project.

The work in Chapter 4 and Chapter 5 was done in collaboration with Tintri. First, I feel very lucky to have had Brandon Salmon as my manager. Brandon is one of the friendliest and most enthusiastic people I know. In addition to his technical expertise, Brandon has a strong understanding of business and how a company operates; I'm grateful for all he taught me in that regard. Second, Rose Liu is one the most helpful people with whom I have worked. It seems she has no notion of an issue being "somebody

else's problem." My first interaction with Rose was after she overhead me describing an issue I was facing while integrating Slacker with VMstore. Rose not only volunteered to help me solve said problem, she learned how the current prototype of Slacker worked at the time and suggested a major redesign that resembles our final published work. Finally, I thank John Schmitt and Sean Chen for using Slacker, providing much useful feedback, and helping me solve multiple issues.

Friends and Peers: I have had wonderful peers at UW-Madison, from whom I have learned a great deal. Thanh Do has some of the best work habits I know, and I have learned much from him. I am also grateful for the life wisdom he has shared. Zev Weiss is an incredibly strong hacker. Under tight deadlines, he remains remarkably calm and composed; he is an excellent person to work with under high pressure. Suli Yang is a very strong mathematical and statistical thinker, which is uncommon in the systems community. Suli is one of my favorite people with whom to debate; she argues passionately for her positions, but will happily change her view in a moment if she discovers another angle. I also thank the following people for giving me the opportunity to work with them: Nishant Agrawal, Samer Al-Kiswany, Brandon Davis, Ben Gellman, Scott Hendrickson, Mike Kellum, Chris Dragga, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Ed Oakes, Stephen Sturdevant, and Michael Vaughn. Finally, I thank the many students of the Arpaci-Dusseau group, and more broadly the students in WISDOM, for creating a supportive community and providing feedback on my work.

Many good friends have been a great support to me during my PhD. First, I thank Scott Larson, Kelly Arneson, Varun Ramesh, and Kristin Haider for our many adventures together. I thank them for their careful belaying: if any of them had dropped the rope while I was climbing, I would not be here to write this dissertation. I thank Nathan Weyenberg and Caitlin Weyenberg for being a support during hard times and for

the many meals they have shared with me. I thank Sarah Christoffersen: graduate students with deadlines are not fun people to date, but she was patient with me through several paper submissions. I thank Andy Dibble for challenging my beliefs and helping me become a better person. I thank Adam Maus and Erkin Otles for helping me optimize my life. Finally, I thank Ishani Ahuja, Bryan Crompton, Brandon Davis, Adam Everspaugh, Cong Han Lim, Ron Osuocha, Sankar Panneerselvam, Jim Paton, Wesley Reardon, Brent Stephens, Kristi Stephens, Mark VanderKooy, Venkat Varadarajan, and Raja Ram Yadhav for their friendship.

An Early Start: For most people, it takes a very long time to become a competent programmer, so I believe those who start young have a distinct advantage. Unfortunately, far too few people are given this advantage, so I am especially grateful to the people who helped give me an early start. First, I would like to thank my dad for building Lego robots with me and for his admirable attempts to learn Visual Basic himself so he could then teach me. When his attempts failed, he found somebody who could program and teach, namely Josiah Plopper. And so second, I would like to thank Josiah for tutoring me and getting me started. The extensive notes he wrote and creative projects he devised gave me a fun and smooth introduction to programming. Third, I would like to thank Scott Finn for the many excellent courses I took with him at Western Technical College. Scott is an incredibly pragmatic instructor and cares deeply about students. He is one of very few instructors I've met who writes code live in class, and from his example I learned how to be an effective programmer.

Family: I would like conclude by thanking my parents and sister, who have always been very supportive of my goals and PhD work. First, I thank my mom for her hard work homeshooling me and giving me an appreciation for education and a love of books. She has always been available to offer encouragement and has sent me many a care package before difficult paper deadlines. Second, I thank my dad for teaching me how to

think like an engineer and for working on many projects with me growing up. In particular, when I was six, he taught me that collecting and plotting the right data is the key to solving even the strangest of problems. Finally, I thank Ashia for being the best sister I could possibly ask for, a close friend, and a source of wisdom. Ashia and her husband Seth make Madison home for me.

Contents

Acknowledgments **ii**

Contents **viii**

Abstract **xii**

1 Introduction **1**
1.1 Libraries: Apple Desktop Applications 6
1.2 Layering: Facebook Messages 8
1.3 Microservices: Docker Containers 10
1.4 Slacker: A Lazy Docker Storage Driver 12
1.5 Contributions and Findings 13
1.6 Overview 15

2 Apple Desktop Measurement **16**
2.1 Background 16
2.1.1 Modularity with Libraries 17
2.1.2 iLife and iWork 19
2.2 Measurement Methodology 22
2.3 Pages Export Case Study 25
2.4 Open Types: Directories and Files 29
2.5 Nature of Files 32

2.5.1 File Types . . . 32
2.5.2 File Sizes . . . 34
2.6 Access Patterns . . . 36
2.6.1 I/O Mechanisms . . . 36
2.6.2 File Accesses . . . 38
2.6.3 Read and Write Sizes . . . 40
2.6.4 Sequentiality . . . 43
2.6.5 Preallocation . . . 46
2.6.6 Open Durations . . . 48
2.6.7 Metadata Access . . . 50
2.7 Memory: Caching, Buffering, and Prefetching . . . 52
2.7.1 Reuse and Overwrites . . . 53
2.7.2 Caching Hints . . . 56
2.8 Transactional Properties . . . 58
2.8.1 Durability . . . 59
2.8.2 Atomic Writes . . . 62
2.8.3 Isolation via File Locks . . . 64
2.9 Threads and Asynchronicity . . . 66
2.10 Summary . . . 69

3 Facebook Messages Measurement 72
3.1 Background . . . 73
3.1.1 Versioned Sparse Tables . . . 73
3.1.2 Messages Architecture . . . 74
3.2 Measurement Methodology . . . 75
3.2.1 Trace Collection and Analysis . . . 75
3.2.2 Modeling and Simulation . . . 77
3.2.3 Sensitivity Analysis . . . 78
3.2.4 Confidentiality . . . 80
3.3 Workload Behavior . . . 80
3.3.1 Multilayer Overview . . . 81

3.3.2 Data Types . . . 83
3.3.3 File Size . . . 85
3.3.4 I/O Patterns . . . 87
3.4 Tiered Storage: Adding Flash . . . 90
3.4.1 Performance without Flash . . . 91
3.4.2 Flash as Cache . . . 93
3.4.3 Flash as Buffer . . . 96
3.4.4 Is Flash Worth the Money? . . . 98
3.5 Layering: Pitfalls and Solutions . . . 100
3.5.1 Layering Background . . . 101
3.5.2 Local Compaction . . . 102
3.5.3 Combined Logging . . . 105
3.6 Summary . . . 107

4 Docker Measurement **109**
4.1 Background . . . 110
4.1.1 Version Control for Containers . . . 110
4.1.2 Storage Driver Interface . . . 112
4.1.3 AUFS Driver Implementation . . . 114
4.2 Measurement Methodology . . . 115
4.3 Image Data . . . 119
4.4 Operation Performance . . . 122
4.5 Layers . . . 124
4.6 Caching . . . 127
4.7 Summary . . . 129

5 Slacker: A Lazy Docker Storage Driver **130**
5.1 Measurement Implications . . . 131
5.2 Architectural Overview . . . 133
5.3 Storage Layers . . . 134
5.4 VMstore Integration . . . 136

5.5 Optimizing Snapshot and Clone . . . 138
5.6 Linux Kernel Modifications . . . 140
5.7 Evaluation . . . 142
5.7.1 HelloBench Workloads . . . 142
5.7.2 Long-Running Performance . . . 144
5.7.3 Caching . . . 144
5.7.4 Scalability . . . 146
5.8 Case Study: MultiMake . . . 147
5.9 Framework Discussion . . . 150
5.10 Summary . . . 150

6 Related Work **152**
6.1 Workload Measurement . . . 152
6.2 Layer Integration . . . 154
6.3 Deployment . . . 156
6.4 Cache Sharing . . . 157

7 Conclusions and Future Work **160**
7.1 Summary . . . 161
7.1.1 Apple Desktop . . . 161
7.1.2 Facebook Messages . . . 162
7.1.3 Docker Containers . . . 163
7.1.4 Slacker . . . 163
7.2 Lessons Learned . . . 164
7.3 Future Work . . . 167
7.4 Closing Words . . . 170

Bibliography **172**

Abstract

The types of data we entrust to our storage systems are wide and diverse, including medical records, genetic information, maps, images, videos, music, sales records, restaurant reviews, messages, documents, presentations, software, and code. These storage workloads have very different I/O patterns and have thus created a need for many different storage systems. Unfortunately, building new storage systems "from scratch" for every new workload is cost prohivitive in terms of engineering effort. In order to cope with this challenge, systems builders typically decompose a storage system into subcomponents and reuse existing subcomponents whenever possible. Unfortunately, this can lead to inneficiency and unpredictability. Subsystems are being used that were not optimized for the storage task at hand, and unexpected behaviors often emerge when subcomponents are combined.

In this disseration, we ask several questions. *What are the storage needs of modern applications? How does modularity and code reuse impact the handling of I/O requests across layers and subsystems?* And finally, *what unexpected I/O behaviors emerge when subsystems are composed?* We explore these questions by performing three measurement studies on Apple desktop applications, Facebook Messages, and Docker containers. We further use the findings from the Docker study to optimize container deployment.

First, we study the I/O patterns of six Apple desktop applications. In

particular, we collect and analyze system-call traces for 34 tasks one might perform while using these applications. In addition to confirming standard past findings (*e.g.*, most files are small; most bytes accessed are from large files [10]), we observe a number of new trends. For example, all of the applications heavily use `fsync` and `rename` to durably and atomically update file data. In most of the tasks we study, the application forces a majority of the data written to the file system to disk. These patterns are especially costly for file systems because the amount of data flushed is small, commonly less than 4 KB. An analysis of user-space call stacks suggests that many of these costly operations originate from general-purpose libraries, and may not correspond with programmer intent.

Second, we present a multilayer study of the Facebook Messages stack, which is based on HBase and HDFS. We collect and analyze HDFS traces to identify potential improvements, which we then evaluate via simulation. Messages represents a new HDFS workload: whereas HDFS was built to store very large files and receive mostly-sequential I/O, 90% of files are smaller than 15 MB and I/O is highly random. We find hot data is too large to easily fit in RAM and cold data is too large to easily fit in flash; however, cost simulations show that adding a small flash tier improves performance more than equivalent spending on RAM or disks. HBase's layered design offers simplicity, but at the cost of performance; our simulations show that network I/O can be halved if compaction bypasses the replication layer. Finally, although Messages is read-dominated, several features of the stack (*i.e.*, logging, compaction, replication, and caching) amplify write I/O, causing writes to dominate disk I/O.

Third, we study the I/O patterns generated by the deployment of Docker containers. Towards this end, we develop a new container benchmark, HelloBench, to evaluate the startup times of 57 different containerized applications. We use HelloBench to analyze workloads in detail, studying the block I/O patterns exhibited during startup and compress-

ibility of container images. Our analysis shows that pulling packages accounts for 76% of container start time, but only 6.4% of that data is read.

Finally, we use our analysis of Docker to guide the design of Slacker, a new Docker storage driver optimized for fast container startup. Slacker is based on centralized storage that is shared between all Docker workers and registries. Workers quickly provision container storage using backend clones and minimize startup latency by lazily fetching container data. Slacker speeds up the median container development cycle by 20× and deployment cycle by 5×.

1 Introduction

Emergent Properties: *"properties that are not evident in the individual components, but they show up when combining those components"*

...

"they might also be called surprises"

– Saltzer and Kaashoek [91]

Our society is largely driven by data. A major data loss is devastating to many companies [57], and many types of data carry immeasurable personal value (*e.g.*, family photos). The storage systems that preserve this data thus represent a core infrastructure that supports us in our daily lives. The types of data we entrust to our storage systems are wide and diverse, including medical records, genetic information, maps, images, videos, music, sales records, restaurant reviews, messages, documents, presentations, software, and code. The patterns with which we access this data are similarly diverse: some data is quickly deleted after being generated; other data is worth keeping forever. Some data is read repeatedly; other data is preserved "just in case" it is someday needed. Some data is scanned sequentially, in order from beginning to end; other data is accessed in random, unpredictable ways.

The diversity of modern storage workloads creates a corresponding need to build many different storage systems. A generic storage system cannot well serve many different applications because a workload's characteristics have many implications for how the underlying storage system should represent and manipulate data. For example, *data lifetime* has implications for how aggressively the storage system should reorganize data in the background. If most data is deleted soon after it is created, compacting or defragmenting data for optimal layout is a waste of I/O resources. *Access skew* has implications for caching and hybrid storage. If most I/O is to a small portion of the total data, the storage system should probably keep the hot data and cold data on different devices that are faster and slower respectively. *Sequentiality* has implications for storage layout. If reads to a file are issued in order, from the beginning to the end of the file, the storage system should similarly arrange the file data in the same order on disk for optimal access.

Unfortunately, storage systems are exceedingly complex and time consuming to design, implement, and operate. Local file systems are frequently a core component of a larger storage system, but local file systems alone are typically 30-75K lines of code [93]. People are continually finding new bugs in these file systems, even those that are most mature [66]. Building a new storage system "from scratch" for every new storage workload is simply not feasible. We thus face a challenge: *how can we build a broad range of storage systems to meet the diverse demands of modern applications without expending inordinate engineering effort doing so?*

The Case for Modularity: The solution has been to modularize and reuse various storage components. Rather than cutting new systems from whole cloth, engineers typically build new systems as a composition of subsystems. This divide-and-conquer approach simplifies reasoning about the storage problem at hand and enables reuse of subsystems in different storage stacks.

Dijkstra found layering (one way to compose subsystems) "proved to be vital for the verification and logical soundness" of the THE operating system [29]. Modularity was also helpful in the Frangipani distributed file system [103]. Construction of Frangipani was greatly simplified because it was implemented atop Petal [62], a distributed and replicated block-level storage system. Because Petal provides scalable, fault-tolerant virtual disks, Frangipani could focus solely on file-system level issues (*e.g.*, locking). The result of this two-layer structure, according to the authors, was that Frangipani was "relatively easy to build" [103].

Modular storage also enables reuse. For example, GFS (the Google File System) was built with a set of workloads at Google in mind [40], such as MapReduce [26]. GFS manages sharding and replication, and thereby tolerates disk and machine failures. When Google later decided to build a distributed database, Bigtable [20], they reused GFS. This decision allowed them to focus on higher-level logic in Bigtable (*e.g.*, how to represent rows and columns) without needing to reimplement all the fault tolerance functionality provided by GFS.

Emergent Properties: Building modular storage systems and reusing components has many advantages, but oftentimes the final system has characteristics one would not expect. These characteristics have been called *emergent properties* because they are not evident in any of the individual components and only emerge when the subcomponents are combined.

Because storage systems are exceedingly complex, and their interactions are even more so, emergent properties are usually "surprises" [91] to developers. Unfortunately, these surprises are rarely pleasant (*e.g.*, one never achieves unexpectedly good scalability after gluing together old components to construct a new storage stack). Instead, to the dismay of many a systems builder, the surprises usually emerge in the form of decreased performance, lowered reliability, and other related issues. Denehy *et al.* show that combining journaling with a software RAID can

lead to data loss or corruption if one is not careful [28]. Others have similarly argued about the general inefficiency of the file system atop block devices [39]. *"The most pernicious and subtle bugs are system bugs arising from mismatched assumptions made by the authors of various components"* [17].

The problem of emergent properties is exacerbated by the fact that most software is too large to be built by individuals or small teams. Even Linus Torvalds is responsible for writing only 2% of the code in his namesake operating system, Linux [86]. The subsystems from which we compose modern storage systems are not co-designed with a single, well-defined storage problem in mind. Brooks argues that conceptual integrity *"dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds"* [17]. Modern storage systems necessarily lack this kind of conceptual integrity. Consequently, unexpected and inefficient I/O behaviors emerge when we deploy these systems.

The Need to Measure: Given that developers can usually only be deeply familiar with a small fraction of the storage stack to which they contribute, *how can we know whether a composed storage system effectively serves a particular workload?* Emergent properties make it impossible to answer this question by design. As Saltzer and Kaashoek advise, *"it is wise ... to focus on an unalterable fact of life: some things turn up only when a system is built"* [91]. If we follow this advice, the best we can do is make sure emergent properties do not remain unknown. Measurement is the key to uncovering emergent properties: once a system is been built, it should be traced and profiled. The resulting measurement findings can then drive the next iteration of building.

One excellent example of measurement-driven storage is found in the development of the Andrew File System [52]. Detailed analysis of an early AFS prototype showed, among other things, that workloads were dominated by `stat` calls used to check for cache staleness. The measurements led to the next-generation protocol, which included the key innovation

of callbacks. Much of the work in storage systems over the past three decades has been similarly driven by measurement, the deep and detailed analysis of I/O workloads [10, 32, 33, 52, 64, 78, 88, 95, 109].

This Dissertation: New applications are constantly being written and the corresponding I/O workloads are always evolving, so the work of measurement is never done. At its core, this dissertation is part of that ongoing work, providing three new data points in the large space of I/O workloads. Our work exposes the inner workings of three types of modern application. In particular, we study the I/O behavior of Apple desktop applications, Facebook Messages, and Docker containers. These applications have previously been the subject of little or no published I/O behavior analysis.

This dissertation is also a study of modularity in storage systems. In addition to reporting basic I/O characteristics (*e.g.*, file sizes and sequentiality), our exploration is attuned to the fact that all our applications are a composition of subsystems. Our three applications represent three composition patterns. The desktop applications rely heavily on user-space libraries. Facebook Messages stores its data is a database (HBase) backed by a distributed file system (HDFS) originally built for very different workloads (*e.g.*, MapReduce jobs). Docker containers are a fundamental building block for decomposing applications into microservices.

Finally, this dissertation is a study of emergent properties. In all three studies, we find I/O behaviors of the composed system that would likely be "surprises" to the developers of these systems. The Apple applications use expensive operations (*e.g.*, `fsync` and `rename`) for relatively unimportant data. Facebook Messages is based on a stack that amplifies writes from 1% of all I/O (at the database level) to 64% (at the device level). Docker containers give every microservice its own file system environment with the result that over 90% of the data copied during deployment is not necessary.

The purpose of measurement is to inform the next iteration of building. Our findings have many implications for how future applications and storage stacks should be built. While we do not implement all of our suggestions, we use our measurement of Docker containers to drive the design of a new Docker storage driver, Slacker. Relative to Docker running with a traditional driver, Slacker improves container startup latency by 5x. In contrast to this large improvement, our changes were relatively small and focused within the Docker codebase. Because our optimizations were guided by analysis, we were able to achieve major gains with very targeted implementation efforts.

The central finding of this dissertation is that *modularity is not free*. In all the applications we consider, we find excessive I/O related to layering and other forms of composition. To qualify our thesis, we are not opposed to modularity; decomposition has many advantages including maintainability, reusability, and isolation. These properties are often valuable enough to justify a performance cost. However, we believe it is crucial to measure workloads and storage systems to identify and mitigate the costs of modularity. Modern software lacks conceptual integrity, but efficiency is not a lost cause. We find that relatively simple measurement-driven adaptations work surprisingly well.

1.1 Libraries: Apple Desktop Applications

Whereas most studies of file systems focus on the corporate or academic intranet, most file-system users work in the more mundane environment of the *home*, accessing data via desktop PCs, laptops, and compact devices such as tablet computers and mobile phones. Despite the large number of previous studies, little is known about home-user applications and their I/O patterns.

Home-user applications are important today, and their importance

will increase as more users store data not only on local devices but also in the cloud. Users expect to run similar applications across desktops, laptops, and phones; therefore, the behavior of these applications will affect virtually every system with which a user interacts. I/O behavior is especially important to understand since it greatly impacts how users perceive overall system latency and application performance [35].

Home-user applications are fundamentally large and complex, containing millions of lines of code [67]. These applications are often developed in sophisticated IDEs and leverage powerful libraries, such as Cocoa and Carbon. Whereas UNIX-style applications often directly invoke system calls to read and write files, modern libraries put more code between applications and the underlying file system; for example, including `"cocoa.h"` in a Mac application imports 112,047 lines of code from 689 different files [83]. A goal of this work is to learn whether I/O is processed across library boundaries in a reasonable and consistent way, or whether conceptual integrity is sacrificed in the design of these applications.

In this study, we present the first in-depth analysis of the I/O behavior of modern home-user applications; we focus on productivity applications (for word processing, spreadsheet manipulation, and presentation creation) and multimedia software (for digital music, movie editing, and photo management). Our analysis centers on two Apple software suites: iWork, consisting of Pages, Numbers, and Keynote; and iLife, which contains iPhoto, iTunes, and iMovie. As Apple's market share grows [104], these applications form the core of an increasingly popular set of workloads; as device convergence continues, similar forms of these applications are likely to access user files from both stationary machines and moving cellular devices. We call our collection the *iBench task suite.*

To investigate the I/O behavior of the iBench suite, we build an instrumentation framework on top of the powerful DTrace tracing system found inside Mac OS X [19]. DTrace allows us not only to monitor system

calls made by each traced application, but also to examine stack traces, in-kernel functions such as page-ins and page-outs, and other details required to ensure accuracy and completeness. We also develop an application harness based on AppleScript [6] to drive each application in the repeatable and automated fashion that is key to any study of GUI-based applications [35].

Our careful study of the tasks in the iBench suite has enabled us to make a number of interesting observations about how applications access and manipulate stored data. In addition to confirming standard past findings (*e.g.*, most files are small; most bytes accessed are from large files [10]), we observe a number of new trends. For example, all of the applications heavily use `fsync` and `rename` to durably and atomically update file data. In most of the tasks we study, the application forces a majority of the data written to the file system to disk. These patterns are especially costly for file systems because the amount of data flushed is small, commonly less than 4 KB. An analysis of user-space call stacks suggests that many of these costly operations originate from general-purpose libraries, and may not correspond with programmer intent.

1.2 Layering: Facebook Messages

In this study, we focus on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from Bigtable [20]) atop a distributed file system (HDFS [97], derived from the Google File System [40]). Our goal is to study the interaction of these important systems, with a particular focus on the lower layer; thus, our highest-level question: is HDFS an effective storage backend for HBase?

To derive insight into this hierarchical system, and thus answer this question, we trace and analyze it under a popular workload: Facebook Messages [72]. Facebook Messages is a messaging system that enables

Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. Facebook Messages stores its information within HBase (and thus, HDFS), and hence serves as an excellent case study.

To perform our analysis, we first collect detailed HDFS-level traces over an eight-day period on a subset of machines within a specially configured *shadow cluster*. Facebook Messages traffic is mirrored to this shadow cluster for the purpose of testing system changes; here, we utilize the shadow to collect detailed HDFS traces. We then analyze said traces, comparing results to previous studies of HDFS under more traditional workloads [47, 56].

To complement to our analysis, we also perform numerous simulations of various caching, logging, and other architectural enhancements and modifications. Through simulation, we can explore a range of "what if?" scenarios, and thus gain deeper insight into the efficacy of the layered storage system.

Overall, we derive numerous insights, some expected and some surprising, from our combined analysis and simulation study. From our analysis, we find writes represent 21% of I/O to HDFS files; however, further investigation reveals the vast majority of writes are HBase overheads from logging and compaction. Aside from these overheads, Facebook Messages writes are scarce, representing only 1% of the "true" HDFS I/O. Diving deeper in the stack, simulations show writes become amplified. Beneath HDFS replication (which triples writes) and OS caching (which absorbs reads), 64% of the final disk load is write I/O. This write blowup (from 1% to 64%) emphasizes the importance of optimizing writes in layered systems, even for especially read-heavy workloads like Facebook Messages.

From our simulations, we further extract the following conclusions. We find that caching at the DataNodes is still (surprisingly) of great util-

ity; even at the last layer of the storage stack, a reasonable amount of memory per node (*e.g.*, 30 GB) significantly reduces read load. We also find that a "no-write allocate" policy generally performs best, and that higher-level hints regarding writes only provide modest gains. Further analysis shows the utility of server-side flash caches (in addition to RAM), *e.g.*, adding a 60 GB SSD can reduce latency by 3.5x.

Finally, we evaluate the effectiveness of more substantial HDFS architectural changes, aimed at improving write handling: local compaction and combined logging. Local compaction performs compaction work within each replicated server instead of reading and writing data across the network; the result is a 2.7x reduction in network I/O. Combined logging consolidates logs from multiple HBase RegionServers into a single stream, thus reducing log-write latencies by 6x.

1.3 Microservices: Docker Containers

In this study, we analyze Docker, a microservice development and deployment tool. Microservice architectures decompose applications into loosely coupled components that control their own environments and dependencies and communicate over well-defined interfaces. This approach improves developer velocity because programmers have more control over the environment in which their component runs, and they need not understand everything about the surrounding components [24].

In a microservice architecture, each microservice is isolated from other microservices. Hypervisors, or virtual machine monitors (VMMs), have traditionally been used to provide isolation for applications [18, 45, 110]. Each application is deployed in its own virtual machine, with its own environment and resources. Unfortunately, hypervisors need to interpose on various privileged operations (*e.g.*, page-table lookups [1, 18]) and use roundabout techniques to infer resource usage (*e.g.*, ballooning [110]).

The result is that hypervisors are heavyweight, with slow boot times [119] as well as run-time overheads [1, 18]. As applications and systems are further decomposed into smaller services, these overheads become increasingly odious.

Fortunately, containers, as driven by the popularity of Docker [70], have recently emerged as a lightweight alternative to hypervisor-based virtualization. Within a container, all process resources are virtualized by the operating system, including network ports and file-system mount points. Containers are essentially just processes that enjoy virtualization of all resources, not just CPU and memory; as such, there is no intrinsic reason container startup should be slower than normal process startup. Thus, containers might appear an ideal abstraction on which to build microservices.

Unfortunately, containers also have overheads and startup delays in practice one might not expect given their similarity to traditional processes. In particular, deployment tools built for containers frequently spend significant time initializing a container when an application runs on a new worker for the first time. Storage initialization is especially expensive. Whereas initialization of network, compute, and memory resources is relatively fast and simple (*e.g.*, zeroing memory pages), a containerized application requires a fully initialized file system, containing application binaries, a complete Linux distribution, and package dependencies. Deploying a container in a Docker or Google Borg [108] cluster typically involves significant copying and installation overheads. A recent study of Google Borg revealed: *"[task startup latency] is highly variable, with the median typically about 25 s. Package installation takes about 80% of the total: one of the known bottlenecks is contention for the local disk where packages are written"* [108].

If the latency of starting a containerized application on a new worker can be reduced, a number of opportunities arise: microservices can scale

instantly to handle flash-crowd events [34], cluster schedulers can frequently rebalance nodes at low cost [50, 108], software upgrades can be rapidly deployed when a security flaw or critical bug is fixed [82], and developers can interactively build and test distributed applications [90].

We study the obstacles to fast container startup by developing a new open-source Docker benchmark, HelloBench, that carefully exercises container startup. HelloBench is based on 57 different container workloads and measures the time from when deployment begins until a container is ready to start doing useful work (*e.g.*, servicing web requests).

We then perform a measurement study on Docker. We use HelloBench and static analysis to characterize container images and I/O patterns. Among other findings, our analysis shows that (1) copying package data accounts for 76% of container startup time, (2) only 6.4% of the copied data is actually needed for containers to begin useful work, and (3) simple block-deduplication across images achieves better compression rates than gzip compression of individual images.

1.4 Slacker: A Lazy Docker Storage Driver

We use the finding of our Docker measurement study to optimize container deployment and startup in Docker. In particular, we construct Slacker, a new Docker storage driver that achieves fast container distribution by utilizing specialized storage-system support at multiple layers of the stack. Slacker uses the snapshot and clone capabilities of our backend storage server (a Tintri VMstore [106]) to dramatically reduce the cost of common Docker operations. Rather than prepropagate whole container images, Slacker lazily pulls image data as necessary, drastically reducing network I/O. Slacker also utilizes modifications we make to the Linux kernel in order to improve cache sharing.

The result of using these techniques is a massive improvement in the

performance of common Docker operations; image pushes become 153× faster and pulls become 72× faster. Common Docker use cases involving these operations greatly benefit. For example, Slacker achieves a 5× median speedup for container deployment cycles and a 20× speedup for development cycles.

We also build MultiMake, a new container-based build tool that showcases the benefits of Slacker's fast startup. MultiMake produces 16 different binaries from the same source code, using different containerized GCC releases. With Slacker, MultiMake experiences a 10× speedup.

1.5 Contributions and Findings

We describe the main contributions of this dissertation:

- We collect and analyze traces of three new sets of applications: Apple desktop applications, Facebook Messages, and Docker containers.
- We publicly released the trace tools we developed for tracing system calls[1] and HDFS workloads[2].
- We publicly release our Apple desktop traces[3]; these have been used in the Magritte benchmark [114].
- We publicly release a new Docker startup benchmark based on 57 workloads, HelloBench[4].
- We build a storage stack simulator to study how Facebook Messages I/O is processed across layers.

[1] `http://research.cs.wisc.edu/adsl/Traces/ibench/utilities.tar.gz`

[2] `https://github.com/facebookarchive/hadoop-20/blob/master/src/hdfs/org/apache/hadoop/hdfs/APITraceFileSystem.java`

[3] `http://research.cs.wisc.edu/adsl/Traces/ibench`

[4] `https://github.com/Tintri/hello-bench`

- We build Slacker, a new Docker storage driver optimized for fast startup; Slacker utilizes Linux kernel modifications and the extended API of a Tintri VMstore.

We draw a number of general conclusions from our three analysis studies, which we describe in more detail in Section 7.2:

Modularity often causes unnecessary I/O. The Apple desktop applications force unimportant data to disk, HBase copies data over the network that workers could compute locally, and Docker copies a significant amount of package data that is not strictly needed. All these behaviors are related to the modularity of the system being studied.

Layers mask costs. A high-level request typically passes through many layers before reaching the underlying device. In all our studies, we found cases where the cost of high-level operations was amplified at lower layers of the stack in ways a developer would likely not expect.

Simple measurement-driven adaptations work surprisingly well. Our simulation of HBase/HDFS integrations and our work on Slacker show that simple optimizations can mitigate the costs of modularity.

Files remain small. We find that most files are small in our analysis, as is found in many prior studies. For the Apple desktop applications, this is unsurprising, but small files represent a new pattern for which HDFS was not originally built.

Cold data is important. Much data is copied and stored that is not actually used in our studies. This finding challenges the assumption that SSDs will soon make disk drives obsolete. It also has highlights the importance of laziness at the software level.

1.6 Overview

We briefly describe the contents of the different chapters in the dissertation.

- **Apple Desktop Measurement**. In Chapter 2, we analyze traces of six Apple Desktop applications. We correlate the system-call invocations with user-space stack traces to understand how libraries and frameworks influence I/O patterns.
- **Facebook Messages Measurement**. In Chapter 3, we analyze traces of Facebook Messages as collected on a shadow cluster. We also use the traces to drive a multilayer simulator we build and explore a number of *what-if* questions related to hybrid storage and caching.
- **Docker Measurement**. In Chapter 4, we describe the construction of our HelloBench benchmark that stresses container startup. We study the images of which HelloBench consists and analyze I/O patterns during the execution of the benchmark.
- **Slacker: A Lazy Docker Storage Driver**. In Chapter 5, we describe a new Docker storage driver we build that lazily fetches container data instead of prefetching everything, as done by other container deployment systems.
- **Related Work**. In Chapter 6, we discuss related measurement work. We also describe various techniques for layer integration, deployment work, and cache sharing strategies.
- **Conclusions and Future Work**. Chapter 7 summarizes our measurement findings and highlights general findings across multiple studies. We also discuss our research with OpenLambda and our plans to explore new ways to build applications with serverless computing.

2
Apple Desktop Measurement

Desktop applications have greatly evolved over the last few decades. In 1974, Ritchie and Thompson wrote *"No large 'access method' routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long"* [87]. Today, desktop applications rely heavily on a variety of libraries for storing and managing data. In this chapter, we analyze six modern desktop applications and study the effects of using complex storage libraries.

The rest of this chapter is organized as follows. We begin by describing how modern applications are developed and the set of applications we select to study (§2.1). Next, we describe our analysis methodology (§2.2). We then look at one workload in detail as a case study (§2.3). Next, we do a broader analysis over six sections, focusing on open patterns (§2.4), file characteristics (§2.5), access patterns (§2.6), memory (§2.7), transactional properties (§2.8), and asynchronicity (§2.9). Finally, we summarize our findings (§2.10).

2.1 Background

Modern desktop and mobile applications rely heavily on libraries for managing and persisting data structures. In this section, we discuss some of

the reasons for choosing to use libraries (§2.1.1) and describe the desktop applications we select for our study (§2.1.2).

2.1.1 Modularity with Libraries

Applications and systems are often divided into subcomponents, or modules, for a variety of reasons, including comprehensibility, reusability, and isolation. The last benefit, isolation, typically entails clear performance costs (*e.g.*, context switching and copying between fault domains). Libraries represent a popular form of modularity without isolation. With libraries, many modules may run in the same fault domain (*e.g.*, a process address space). Like other types of modules, libraries often expose well-defined interfaces and hide internal implementation details. Unlike other types of modules, libraries can easily and efficiently share data with other libraries without copying, but a bug in one library could easily cause the entire process to crash.

Both approaches to decomposing applications (*i.e.*, into processes or libraries) have a long history in application development [79, 87]. Many features of Unix specifically support decomposition into processes. All processes have common I/O interfaces (*e.g.*, stdin and stdout), and pipes enable arbitrary composition of multiple processes. These traditional Unix-based applications are designed to be simple and perform one task well. The user has significant power and flexibility to string these simple programs together to perform more complex tasks.

The Unix approach to decomposition is still in use by some modern users, particularly advanced users (*e.g.*, developers and system administrators), but Unix-style decomposition has not prevailed more broadly [60]. Modern home and office applications are standalone monoliths, providing a rich and continuously evolving set of features to demanding users. These applications are fundamentally large and complex, containing millions of lines of code [67], and relying heavily on many different libraries.

Why have libraries won over processes as the means of modularity for desktop applications? One likely reason is that as more people have become computer users, user sophistication has decreased. Most users want a GUI-based application that intuitively guides them in their work; they do not want to think about how to compose a pipeline of simple processes to accomplish the task at hand. Developers, not users, are now typically responsible for reasoning about how to compose subcomponents to achieve complex goals.

There are also more technical arguments for why applications should be composed from libraries rather than subprocesses. Parnas [79] shows that there are multiple ways to decompose a large task into modules. One way is to decompose around functions, or stages of a pipeline. With this type of decomposition, the UNIX approach is quite natural. However, a second way is to decompose around data structures. This second way allows flexibility of representation (*e.g.*, where and how data is stored), but abstracting data structures is a cross-cutting proposition. A change in representation may affect every stage of a pipelined task. When developers reason about applications in terms of data structures (instead of, for example, functionality or stages of a pipeline), libraries are a more natural form of modularity, because changing code in one library can affect every stage of a pipeline, and copying is unnecessary between stages.

Parnas [79] argues that this type of maintainability is possible due to *information hiding*: the internal representation of data is hidden behind well-defined interfaces, so the many functions interacting with a data structure need not change when the representation changes.

Modern applications rely heavily on libraries for managing data; these libraries hide internal representation details. The use of libraries for data management is even being adopted for structured data stored in databases, which have traditionally been managed as separate processes. Embedded databases (*i.e.*, databases that are built as a library to be used in the same

process address space as the client) are being used when developers want to abstract data representation but do not want to pay the performance cost of inter-process communication [15]. SQLite is a particularly popular embedded database used by many modern applications [58]. SQLite is also used in several of the tasks we study in this chapter, and we will be analyzing its impact on I/O in more detail. Applications also use libraries for less-structured data. For example, all the applications we study rely heavily on an Apple library (*i.e.*, the PList library) for storing key-value data and managing persistence properties. We will show how decisions made by the developers of the PList library greatly influence the I/O behavior of nearly every workload we consider.

Given that libraries can avoid copying between modules and switching costs, one might expect that applications composed of libraries do not experience the modularity-related overheads that are often experienced by applications based on other types of modules. Many of results we present in this chapter will refute this hypothesis. Even when decomposition of program logic is merely logical, as with libraries, modularity often has performance costs due to information hiding. Abstraction has many benefits, but it is not free in terms of performance, even with libraries.

2.1.2 iLife and iWork

In this section, we describe the workloads we study. For our study, we selected six popular Apple desktop applications: iPhoto, iTunes, and iMovie (of the iLife suite), and Pages, Numbers, and Keynote (of the iWork suite). Unfortunately, the research community does not have data on the exact ways home users utilize these applications, so we study a variety of tasks in each application we believe a reasonable user would be likely to perform. Fortunately, our results show many consistent patterns across this range of tasks.

We call our collection of application tasks the iBench task suite. The

Task	Description
iPhoto Start	Open iPhoto with a library of 400 photos
iPhoto Imp	Import 400 photos into an empty library
iPhoto Dup	Duplicate 400 photos from library
iPhoto Edit	Sequentially edit 400 photos
iPhoto Del	Sequentially delete 400 photos
iPhoto View	Sequentially view 400 photos
iTunes Start	Open iTunes with 10 song album
iTunes ImpS	Import 10 song album to library
iTunes ImpM	Import 3 minute movie to library
iTunes PlayS	Play album of 10 songs
iTunes PlayM	Play 3 minute movie
iMovie Start	Open iMovie with 3 minute clip in project
iMovie Imp	Import 3 minute .m4v (20MB) to "Events"
iMovie Add	Paste 3 minute clip from "Events" to project
iMovie Exp	Export 3 minute video clip
Pages Start	Open Pages
Pages New	Create 15 text-page document; save as .pages
Pages NewP	Create 15 JPG document; save as .pages
Pages Open	Open 15 text-page document
Pages PDF	Export 15 page document as .pdf
Pages PDFP	Export 15 JPG document as .pdf
Pages DOC	Export 15 page document as .doc
Pages DOCP	Export 15 JPG document as .doc
Numbers Start	Open Numbers
Numbers New	Save 5 sheets/column graphs as .numbers
Numbers Open	Open 5 sheet spreadsheet
Numbers XLS	Export 5 sheets/column graphs as .xls
Keynote Start	Open Keynote
Keynote New	Create 20 text slides; save as .key
Keynote NewP	Create 20 JPG slides; save as .key
Keynote Play	Open/play presentation of 20 text slides
Keynote PlayP	Open/play presentation of 20 JPG slides
Keynote PPT	Export 20 text slides as .ppt
Keynote PPTP	Export 20 JPG slides as .ppt

Table 2.1: **iBench Task Suite.** *Each of the 34 tasks is briefly described.*

suite contains 34 different task in total; a brief overview is given in Table 2.1. We now describe the six applications in more detail.

iLife iPhoto 8.1.1 (419): a digital photo album and photo manipulation application. iPhoto stores photos in a library that contains the data for the photos (which can be in a variety of formats, including JPG, TIFF, and PNG), a directory of modified files, a directory of scaled down images, and two files of thumbnail images. The library stores metadata in a SQLite database. iBench contains six tasks exercising user actions typical for iPhoto: starting the application and importing, duplicating, editing, viewing, and deleting photos in the library. These tasks modify both the image files and the underlying database. Each of the iPhoto tasks operates on 400 2.5 MB photos, representing a user who has imported 12 megapixel photos (2.5 MB each) from a full 1 GB flash card from a digital camera.

iLife iTunes 9.0.3 (15): a media player capable of both audio and video playback. iTunes organizes its files in a private library and supports most common music formats (*e.g.*, MP3, AIFF, WAVE, AAC, and MPEG-4). iTunes does not employ a database, keeping media metadata and playlists in both a binary and an XML file. iBench contains five tasks for iTunes: starting iTunes, importing and playing an album of MP3 songs, and importing and playing an MPEG-4 movie. Importing requires copying files into the library directory and, for music, analyzing each song file for gapless playback. The music tasks operate over an album (or playlist) of ten songs while the movie tasks use a single 3-minute movie.

iLife iMovie 8.0.5 (820): a video editing application. iMovie stores its data in a library that contains directories for raw footage and projects and files containing video footage thumbnails. iMovie supports both MPEG-4 and QuickTime files. iBench contains four tasks for iMovie: starting iMovie, importing an MPEG-4 movie, adding a clip from this movie into a project, and exporting a project to MPEG-4. The tasks all use a 3-minute movie because this is a typical length for home videos on video-sharing

websites.

iWork Pages 4.0.3 (766): a word processor. Pages uses a ZIP-based file format and can export to DOC, PDF, RTF, and basic text. iBench includes eight tasks for Pages: starting up, creating and saving, opening, and exporting documents with and without images and with different formats. The tasks use 15 page documents.

iWork Numbers 2.0.3 (332): a spreadsheet application. Numbers organizes its files with a ZIP-based format and exports to XLS and PDF. The four iBench tasks for Numbers include starting Numbers, generating a spreadsheet and saving it, opening the spreadsheet, and exporting a spreadsheet to XLS. To model a possible user working on a budget, tasks utilize a five page spreadsheet with one column graph per sheet.

iWork Keynote 5.0.3 (791): a presentation and slideshow application. Keynote saves to a .key ZIP-based format and exports to Microsoft's PPT format. The seven iBench tasks for Keynote include starting Keynote, creating slides with and without images, opening and playing presentations, and exporting to PPT. Each Keynote task uses a 20-slide presentation.

2.2 Measurement Methodology

In the previous section, we described the 34 tasks of our study. In this section, we describe our tracing and analysis methodology. One difficulty of studying home-user applications is that users interact with these application via a GUI, so the iBench tasks are inherently difficult to drive via traditional scripts. Thus, to save time and ensure reproducible results, we automate the tasks via AppleScript, a general-purpose GUI scripting language. AppleScript provides generic commands to emulate mouse clicks through menus and application-specific commands to capture higher-level operations. Application-specific commands bypass a small amount of I/O by skipping dialog boxes; however, we use them whenever possi-

	Name	Description	Files	(MB)	Acc.	(MB)	RD%	WR%	Acc./s	MB/s
iLife iPhoto	Start	Open iPhoto with library of 400 photos	779	(336.7)	828	(25.4)	78.8	21.2	**151.1**	4.6
	Imp	Import 400 photos into empty library	5900	(1966.9)	8709	(3940.3)	74.4	25.6	26.7	**12.1**
	Dup	Duplicate 400 photos from library	2928	(1963.9)	5736	(2076.2)	52.4	47.6	**237.9**	**86.1**
	Edit	Sequentially edit 400 photos from library	12119	(4646.7)	18927	(12182.9)	69.8	30.2	19.6	**12.6**
	Del	Sequentially del. 400 photos; empty trash	15246	(23.0)	15247	(25.0)	21.8	78.2	**280.9**	0.5
	View	Sequentially view 400 photos	2929	(1006.4)	3347	(1005.0)	98.1	1.9	24.1	**7.2**
iLife iTunes	Start	Open iTunes with 10 song album	143	(184.4)	195	(9.3)	54.7	45.3	**72.4**	3.4
	ImpS	Import 10 song album to library	68	(204.9)	139	(264.5)	66.3	33.7	**75.2**	**143.1**
	ImpM	Import 3 minute movie to library	41	(67.4)	57	(42.9)	48.0	52.0	**152.4**	**114.6**
	PlayS	Play album of 10 songs	61	(103.6)	80	(90.9)	96.9	3.1	0.4	0.5
	PlayM	Play 3 minute movie	56	(77.9)	69	(32.0)	92.3	7.7	2.2	1.0
iLife iMovie	Start	Open iMovie with 3 minute. clip in project	433	(223.3)	786	(29.4)	99.9	0.1	**134.8**	**5.0**
	Imp	Import 3 minute .m4v (20MB) to "Events"	184	(440.1)	383	(122.3)	55.6	44.4	29.3	**9.3**
	Add	Paste 3 min. clip from "Events" to project	210	(58.3)	547	(2.2)	47.8	52.2	**357.8**	1.4
	Exp	Export 3 minute video clip	70	(157.9)	546	(229.9)	55.1	44.9	2.3	1.0
iWork Pages	Start	Open Pages	218	(183.7)	228	(2.3)	99.9	0.1	**97.7**	1.0
	New	Create 15 text-page doc; save as .pages	135	(1.6)	157	(1.0)	73.3	26.7	**50.8**	0.3
	NewP	Create 15 JPG doc; save as .pages	408	(112.0)	997	(180.9)	60.7	39.3	**54.6**	**9.9**
	Open	Open 15 text page document	103	(0.8)	109	(0.6)	99.5	0.5	**57.6**	0.3
	PDF	Export 15 page document as .pdf	107	(1.5)	115	(0.9)	91.0	9.0	**41.3**	0.3
	PDFP	Export 15 JPG document as .pdf	404	(77.4)	965	(110.9)	67.4	32.6	**49.7**	**5.7**
	DOC	Export 15 page document as .doc	112	(1.0)	121	(1.0)	87.9	12.1	**44.4**	0.4
	DOCP	Export 15 JPG document as .doc	385	(111.3)	952	(183.8)	61.1	38.9	**46.3**	**8.9**
iWork Numbers	Start	Open Numbers	283	(179.9)	360	(2.6)	99.6	0.4	**115.5**	0.8
	New	Save 5 sheets/col graphs as .numbers	269	(4.9)	313	(2.8)	90.7	9.3	9.6	0.1
	Open	Open 5 sheet spreadsheet	119	(1.3)	137	(1.3)	99.8	0.2	**48.7**	0.5
	XLS	Export 5 sheets/column graphs as .xls	236	(4.6)	272	(2.7)	94.9	5.1	8.5	0.1
iWork Keynote	Start	Open Keynote	517	(183.0)	681	(1.1)	99.8	0.2	**229.8**	0.4
	New	Create 20 text slides; save as .key	637	(12.1)	863	(5.4)	92.4	7.6	**129.1**	0.8
	NewP	Create 20 JPG slides; save as .key	654	(92.9)	901	(103.3)	66.8	33.2	**70.8**	**8.1**
	Play	Open/play presentation of 20 text slides	318	(11.5)	385	(4.9)	99.8	0.2	**95.0**	1.2
	PlayP	Open/play presentation of 20 JPG slides	321	(45.4)	388	(55.7)	69.6	30.4	**72.4**	**10.4**
	PPT	Export 20 text slides as .ppt	685	(12.8)	918	(10.1)	78.8	21.2	**115.2**	1.3
	PPTP	Export 20 JPG slides as .ppt	723	(110.6)	996	(124.6)	57.6	42.4	**61.0**	**7.6**

Table 2.2: **34 Tasks of the iBench Suite.** *The table summarizes the 34 tasks of iBench, specifying the application, a short name for the task, and a longer description of the actions modeled. The I/O is characterized according to the number of files read or written, the sum of the maximum sizes of all accessed files, the number of file accesses that read or write data (Acc.), the number of bytes read or written, the percentage of I/O bytes that are part of a read (or write; RD% and WR%, respectively), and the rate of I/O per CPU-second in terms of both file accesses and bytes (Acc./s and MB/s, respectively). Each core is counted individually, so at most 2 CPU-seconds can be counted per second on our dual-core test machine. CPU utilization is measured with the UNIX* `top` *utility, which in rare cases produces anomalous CPU utilization snapshots; those values are ignored.*

ble for expediency.

We then collect system-call traces using DTrace [19] while the tasks are run by the AppleScript task driver. DTrace is a kernel and user level dynamic instrumentation tool that can be used to instrument the entry and exit points of all system calls dealing with the file system. DTrace also records the current state of the system and the parameters passed to and returned from each call. In addition to collecting I/O traces, we recorded important details about the initial state of the system before task execution (*e.g.*, directory structure and file sizes). Such initial-state snapshots are critical in file-system benchmarking [2] and make it possible to replay the traces [114].

While tracing with DTrace was generally straightforward, we faced four challenges in collecting the iBench traces. First, file sizes are not always available to DTrace; thus, we record every file's initial size and compute subsequent file size changes caused by system calls such as `write` or `ftruncate`. Second, iTunes uses the `ptrace` system call to disable tracing; we circumvent this block by using `gdb` to insert a breakpoint that automatically returns without calling `ptrace`. Third, the `volfs` pseudo-file system in HFS+ (Hierarchical File System) allows files to be opened via their inode number instead of a file name; to include pathnames in the trace, we instrument the `build_path` function to obtain the full path when the task is run. Fourth, tracing system calls misses I/O resulting from memory-mapped files; therefore, we purged memory and instrumented kernel page-in functions to measure the amount of memory-mapped file activity. We found that the amount of memory-mapped I/O is negligible in most tasks; we thus do not include this I/O in the iBench traces or analysis, except for a brief discussion in Section 2.6.1.

All measurements were performed on a Mac Mini running Mac OS X Snow Leopard version 10.6.2 and the HFS+ file system. The machine has 2 GB of memory and a 2.26 GHz Intel Core Duo processor. Our traces are

online at `http://www.cs.wisc.edu/adsl/Traces/ibench`.

Table 2.2 contains a brief description and basic I/O characteristics for each of our 34 iBench tasks. The table illustrates that the iBench tasks perform a significant amount of I/O. Most tasks access hundreds of files, which in aggregate contain tens or hundreds of megabytes of data. The tasks perform widely differing amounts of I/O, from less than a megabyte to more than a gigabyte. Most tasks perform many more reads than writes. Finally, the tasks exhibit high I/O throughput, often transferring tens of megabytes of data per CPU-second. These results show that the iBench task suite represents an interesting I/O workload (*i.e.*, it is not simply CPU bound).

2.3 Pages Export Case Study

The I/O characteristics of modern home-user applications are distinct from those of UNIX applications studied in the past. To motivate the need for a new study, we investigate the complex I/O behavior of a single representative task: the "Pages DOCP" iBench task. In this workload, the Pages (4.0.3) word processor creates a blank document, inserts 15 JPEG images (each of size 2.5 MB), and saves the document as a Microsoft .doc file.

Figure 2.1 shows the I/O this task performs. The top portion of the figure illustrates the accesses performed over the full lifetime of the task: at a high level, it shows that more than 385 files spanning six different categories are accessed by eleven different threads, with many intervening calls to `fsync` and `rename`. The bottom portion of the figure magnifies a short time interval, showing the reads and writes performed by a single thread accessing the primary .doc productivity file. From this one experiment, we make seven observations. We first focus on the single access that saves the user's document (bottom), and then consider the broader

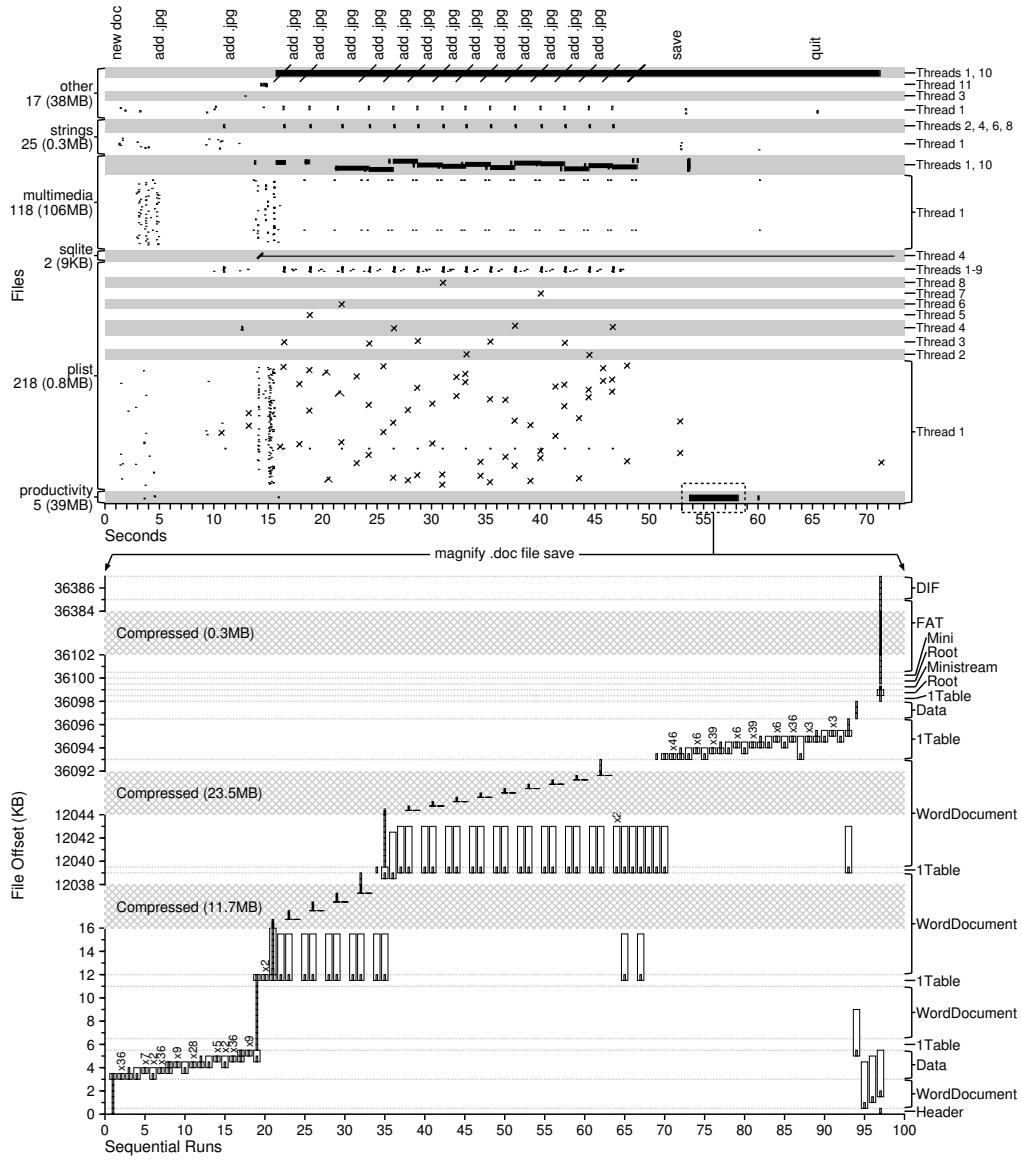

Figure 2.1: **Pages Saving A Word Document.** *Top: All accessed files are shown. Black bars are opens followed by I/O, with thickness logarithmically proportional to bytes of I/O. / is an* `fsync`*; \ is a* `rename`*;* `X` *is both. Bottom: Individual I/O requests to the .doc file during 54-58 seconds are shown. Vertical bar position and bar length represent the offset within the file and number of bytes touched. Thick white bars are reads; thin gray bars are writes. Repeated runs are marked with the number of repetitions. Annotations on the right indicate the name of each file section.*

context surrounding this file save, where we observe a flurry of accesses to hundreds of helper files (top).

A file is not a file. Focusing on the magnified timeline of reads and writes to the productivity .doc file, we see that the file format comprises more than just a simple file. Microsoft .doc files are based on the FAT file system and allow bundling of multiple files in the single .doc file. This .doc file contains a directory (Root), three streams for large data (WordDocument, Data, and 1Table), and a stream for small data (Ministream). Space is allocated in the file with three sections: a file allocation table (FAT), a double-indirect FAT (DIF) region, and a ministream allocation region (Mini).

Sequential access is not sequential. The complex FAT-based file format causes random access patterns in several ways: first, the header is updated at the beginning and end of the magnified access; second, data from individual streams is fragmented throughout the file; and third, the 1Table stream is updated before and after each image is appended to the WordDocument stream.

Auxiliary files dominate. Although saving the single .doc we have been considering is the sole purpose of this task, we now turn our attention to the top timeline and see that 385 different files are accessed. There are several reasons for this multitude of files. First, Pages provides a rich graphical experience involving many images and other forms of multimedia; together with the 15 inserted JPEGs, this requires 118 multimedia files. Second, users want to use Pages in their native language, so application text is not hard-coded into the executable but is instead stored in 25 different .strings files. Third, to save user preferences and other metadata, Pages uses a SQLite database (2 files) and a number of key-value stores (218 .plist files).

Writes are often forced. Pages heavily uses `fsync` to flush write data to disk, making it durable. Pages primarily calls `fsync` on three types of

data. First, Pages regularly uses `fsync` and when updating the key-value store of a .plist file. Second, `fsync` is used on the SQLite database. Third, for each of the 15 image insertions, Pages calls `fsync` on a file named "tempData" (classified as "other") to update its automatic backup.

Renaming is popular. Pages also heavily uses `rename` to atomically replace old files with new files so that a file never contains inconsistent data. This is mostly done in conjunction with calls to `fsync` on .plist files.

Multiple threads perform I/O. Pages is a multi-threaded application and issues I/O requests from many different threads during the experiment. Using multiple threads for I/O allows Pages to avoid blocking while I/O requests are outstanding. Examining the I/O behavior across threads, we see that Thread 1 performs the most significant portion of I/O, but ten other threads are also involved. In most cases, a single thread exclusively accesses a file, but it is not uncommon for multiple threads to share a file.

Frameworks influence I/O. Pages was developed in a rich programming environment where frameworks such as Cocoa or Carbon are used for I/O; these libraries impact I/O patterns in ways the developer might not expect. For example, although the application developers did not bother to use `fsync` or `rename` when saving the user's work in the .doc file, the Cocoa library regularly uses these calls to atomically and durably update relatively unimportant metadata, such as "recently opened" lists stored in .plist files. As another example, when Pages tries to read data in 512-byte chunks from the .doc, each read goes through the `STDIO` library, which only reads in 4 KB chunks. Thus, when Pages attempts to read one chunk from the 1Table stream, seven unrequested chunks from the WordDocument stream are also incidentally read (offset 12039 KB). In other cases, regions of the .doc file are repeatedly accessed unnecessarily. For example, around the 3 KB offset, read/write pairs occur dozens of times. Pages uses a library to write 2-byte words; each time a word is

written, the library reads, updates, and writes back an entire 512-byte chunk. Finally, we see evidence of redundancy between libraries: even though Pages has a backing SQLite database for some of its properties, it also uses .plist files, which function across Apple applications as generic property stores.

This one detailed experiment has shed light on a number of interesting I/O behaviors that indicate that home-user applications are indeed different than traditional workloads. Throughout the remainder of this chapter, we will analyze all 34 tasks of the iBench suite to learn to what degree the patterns observed in this case study apply to Apple desktop applications more generally.

2.4 Open Types: Directories and Files

Files and directories are the two most important objects stored by a file system. We begin our analysis of the iBench task suite by measuring the frequency of access to these two types of object. Applications can use `open` to create a file descriptor corresponding to a file or directory and then use that descriptor to perform I/O. We categorize opens into four groups based on their target and their usage: opens of files that access the files' contents, opens of files that do not access the contents of the files directly, opens of directories that read their contents, and opens of directories that do not read them. We display our results in Figure 2.2.

Our results show that opens to files are generally more common than directory opens, though significant variance exists between tasks. In particular, iMovie Add uses over 90% of its opens for file I/O, while Pages Start uses less than 20% of them for this purpose. Opens to files that access data outnumber those that do not across all tasks; only a few of the iPhoto and iTunes tasks open more than 10-15% of their files without accessing them. Though some opens that do not access files result in no

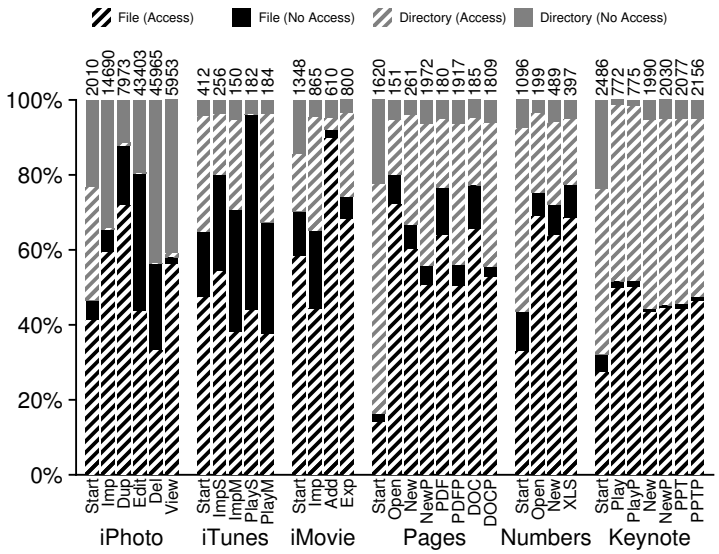

Figure 2.2: **Types of Opens.** *This plot divides opens into four types–file opens that access the file's data, file opens that do not, directory opens that read the directory, and directory opens that do not–and displays the relative size of each. The numbers atop the bars indicate the total number of opens in each task.*

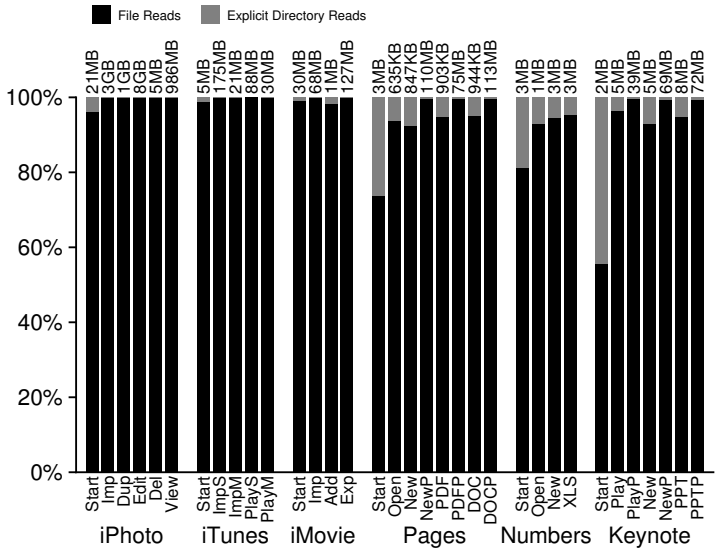

Figure 2.3: **Distribution of Bytes Between File and Directory Reads.** *This plot shows how bytes read are distributed between files and explicit reads of directories. The numbers atop the bars indicate the total number of bytes read.*

further manipulation of their target file, investigating our traces shows that many of these opens are used for locking, calls to `fsync` or `mmap`, and metadata access.

While directory opens usually occur less frequently than file opens, they nonetheless have a significant presence, particularly for iWork. Their quantity varies widely, ranging from over 60% for Keynote Start to under 5% for iTunes PlayS. As with file opens, the majority of directory opens access directory entries, although all iPhoto workloads other than Start access very little data from the directories they open. Similarly, directory opens that do not explicitly access directory entries are not necessarily useless or used solely to confirm the existence of a given directory; many examine metadata attributes of their children or of the directory itself or change the working directory.

Given that a sizable minority of opens are on directories, we next examine how many bytes are read from each type of object in Figure 2.3. This plot omits the implicit directory reads that occur during pathname resolution when files are opened.

Despite the prevalence of directory opens, directory reads account for almost none of the I/O performed in the iLife suite and usually comprise at most 5-7% of the total I/O in iLife. The only exceptions are the Start tasks for Keynote, Pages, and Numbers, where directory reads account for roughly 50%, 25%, and 20%, respectively, of all data read. None of these tasks read much data, however, and, in general, the proportion of data reads from directories diminishes greatly as the total data read for the task rises.

Conclusion: While the iBench applications frequently open directories, the vast majority of reads are from files; the total data read from directories in any given task never exceeds 1 MB. Thus, we focus on file I/O for the rest of our analysis.

2.5 Nature of Files

We now characterize the high-level behavior of the iBench tasks. In particular, we study the different types and sizes of files opened by each iBench task.

2.5.1 File Types

The iLife and iWork applications store data across a variety of files in a number of different formats; for example, iLife applications tend to store their data in libraries (or data directories) unique to each user, while iWork applications organize their documents in proprietary ZIP-based files. The extent to which tasks access different types of files greatly influences their I/O behavior.

To understand accesses to different file types, we place each file into one of six categories, based on file name extensions and usage. *Multimedia* files contain images (*e.g.*, JPEG), songs (*e.g.*, MP3, AIFF), and movies (*e.g.*, MPEG-4). *Productivity* files are documents (*e.g.*, .pages, DOC, PDF), spreadsheets (*e.g.*, .numbers, XLS), and presentations (*e.g.*, .key, PPT). *SQLite* files are database files. *Plist* files are XML property-list files containing key-value pairs for user preferences and application properties. *Strings* files contain strings for localization of application text. Finally, *Other* contains miscellaneous files such as plain text, logs, files without extensions, and binary files.

Figure 2.4 shows the frequencies with which tasks open and access files of each type; most tasks perform hundreds of these accesses. Multimedia file opens are common in all workloads, though they seldom predominate, even in the multimedia-heavy iLife applications. Conversely, opens of productivity files are rare, even in iWork applications that use them; this is likely because most iWork tasks create or view a single productivity file. Because .plist files act as generic helper files, they are rel-

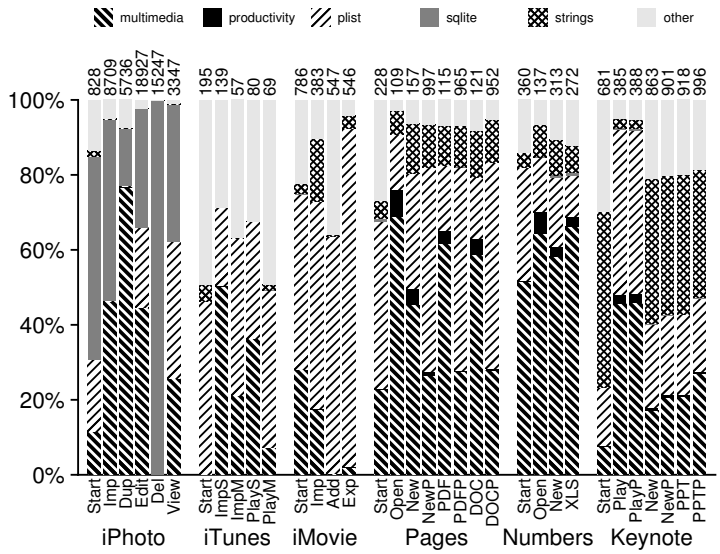

Figure 2.4: **Types of Files Accessed By Number of Opens.** *This plot shows the relative frequency with which file descriptors are opened upon different file types. The number at the end of each bar indicates the total number of unique file descriptors opened on files.*

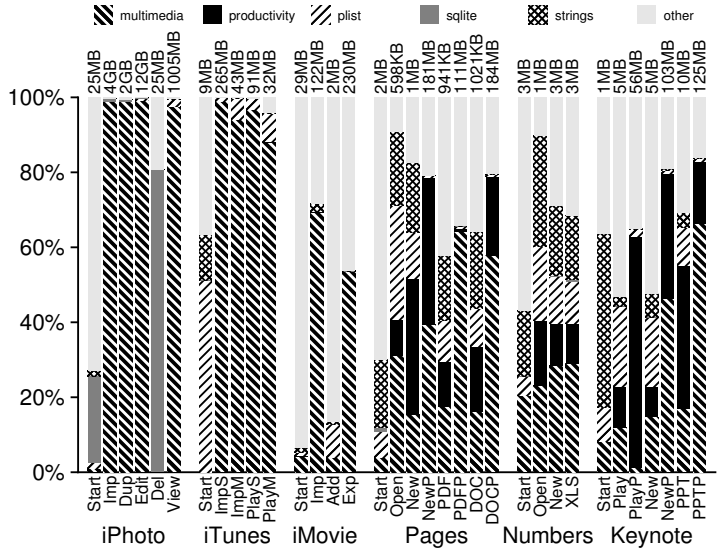

Figure 2.5: **Types of Files Opened By I/O Size.** *This plot shows the relative frequency with which each task performs I/O upon different file types. The number atop each bar indicates the total bytes of I/O accessed.*

atively common. SQLite files only have a noticeable presence in iPhoto, where they account for a substantial portion of the observed opens. Strings files occupy a significant minority of iWork files. Finally, between 5% and 20% of files are of type "Other" (except for iTunes, where they are more prevalent).

Figure 2.5 displays the percentage of I/O bytes accessed for each file type. In bytes, multimedia I/O dominates most of the iLife tasks, while productivity I/O has a significant presence in the iWork tasks; file descriptors on multimedia and productivity files tend to receive large amounts of I/O. SQLite, Plist, and Strings files have a smaller share of the total I/O in bytes relative to the number of opened files; this implies that tasks access only a small quantity of data for each of these files opened (*e.g.*, several key-value pairs in a .plist). In most tasks, files classified as "Other" receive a significant portion of the I/O.

Conclusion: Home applications access a wide variety of file types, generally opening multimedia files the most frequently. iLife tasks tend to access bytes primarily from multimedia or files classified as "Other"; iWork tasks access bytes from a broader range of file types, with some emphasis on productivity files.

2.5.2 File Sizes

Large and small files present distinct challenges to the file system. For large files, finding contiguous space can be difficult, while for small files, minimizing initial seek time is more important. We investigate two questions regarding file size. First, what is the distribution of file sizes accessed by each task? Second, what portion of accessed bytes resides in files of various sizes?

To answer these questions, we record file sizes when each unique file descriptor is closed. We categorize sizes as very small ($<$ 4 KB), small ($<$ 64 KB), medium ($<$ 1 MB), large ($<$ 10 MB), or very large ($\geqslant$ 10 MB).

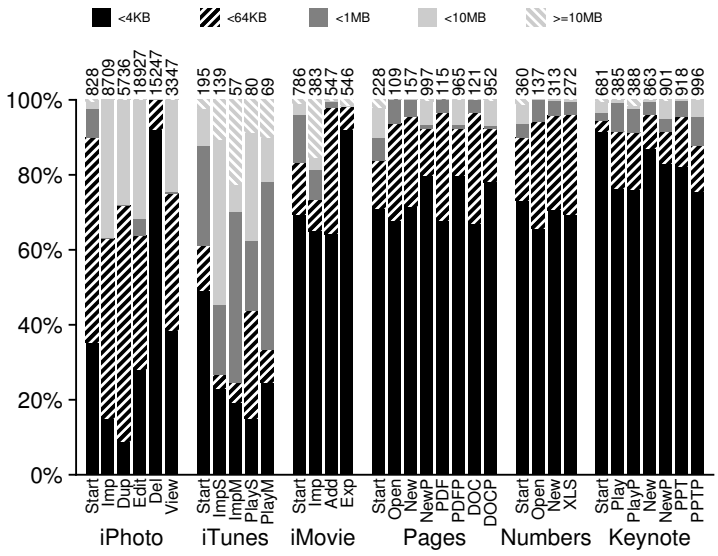

Figure 2.6: **File Sizes, Weighted by Number of Accesses.** *This graph shows the number of accessed files in each file size range upon access ends. The total number of file accesses appears at the end of the bars. Note that repeatedly-accessed files are counted multiple times, and entire file sizes are counted even upon partial file accesses.*

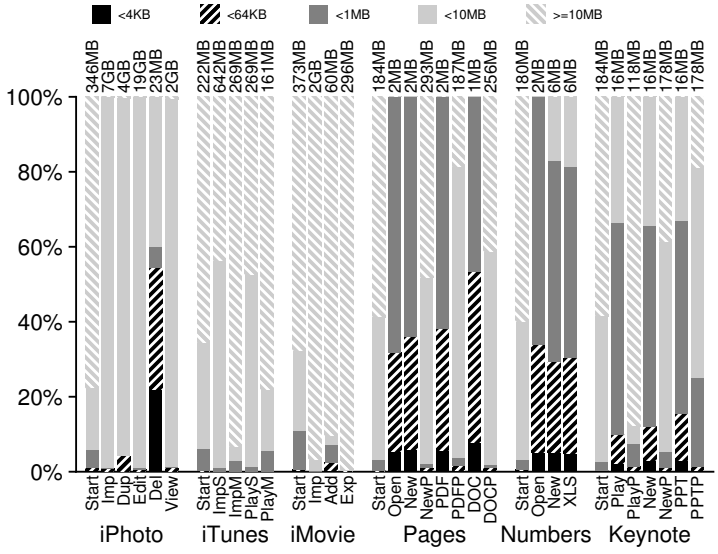

Figure 2.7: **File Sizes, Weighted by the Bytes in Accessed Files.** *This graph shows the portion of bytes in accessed files of each size range upon access ends. The sum of the file sizes appears at the end of the bars. This number differs from total file footprint since files change size over time and repeatedly accessed file are counted multiple times.*

We track how many accesses are to files in each category and how many of the bytes belong to files in each category.

Figure 2.6 shows the number of accesses to files of each size. Accesses to very small files are extremely common, especially for iWork, accounting for over half of all the accesses in every iWork task. Small file accesses have a significant presence in the iLife tasks. The large quantity of small and very small files is due to frequent use of .plist files that store preferences, settings, and other application data; these files often fill just one or two 4 KB pages.

Figure 2.7 shows the proportion of the files in which the bytes of accessed files reside. Large and very large files dominate every startup workload and nearly every task that processes multimedia files. Small files account for few bytes and very small files are essentially negligible.

Conclusion: Agreeing with many previous studies (e.g., [10]), we find that while applications tend to open many very small files ($<$ 4 KB), most of the bytes accessed are in large files ($>$ 1 MB).

2.6 Access Patterns

We next examine how the nature of file accesses has changed, studying the read and write patterns of home applications. These patterns include whether data is transferred via memory-mapped I/O or through read and write requests; whether files are used for reading, writing, or both; whether files are accessed sequentially or randomly; and finally, whether or not blocks are preallocated via hints to the file system.

2.6.1 I/O Mechanisms

UNIX provides two mechanisms for reading and writing to a file that has been opened. First, calls to `read`, `write`, or similar functions may be performed on a file descriptor (we call this request-based I/O). Second, a

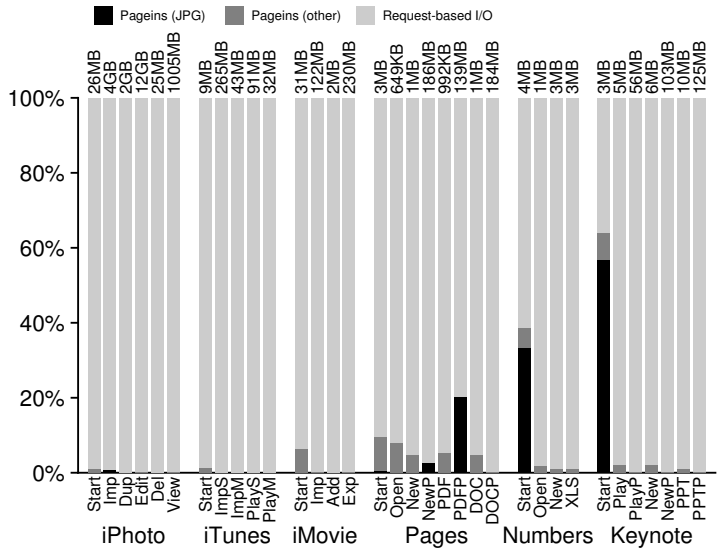

Figure 2.8: **File Access Mechanisms.** *Memory-mapped I/O is compared to request-based I/O. The values atop the bars indicate the sum of both I/O types. Memory-mapped I/O to JPG files is indicated by the black bars. Note that other figures in this chapter exclude memory-mapped I/O.*

process may use `mmap` to map a file into a region of virtual memory and then just access that region. We explore how often these two mechanisms are used.

For each of our tasks, we clear the system's disk cache using the `purge` command so that we can observe all pageins that result from memory-mapped I/O. We are interested in application behavior, not the I/O that results from loading the application, so we exclude pageins to executable memory or from files under /System/Library and /Library. We also exclude pageins from a 13 MB file named "/usr/share/icu/icudt40l.dat"; all the applications use this file, so it should typically be resident in memory. Figure 2.8 shows how frequently pageins occur relative to request-based I/O. We do not observe any pageouts. In general, memory-mapped I/O is negligible. For the two exceptions, Numbers Start and Keynote

Start, the pagein traffic is from JPG files.

In the Numbers and Keynote Start tasks, the user is presented with a list of templates from which to choose. A JPG thumbnail represents each template. The applications map these thumbnails into virtual memory.

Conclusion: The vast majority of I/O is performed by reading and writing to open file descriptors. Only a few of the iBench tasks have significant pageins from memory-mapped files; most of this pagein traffic is from images. For the rest of our analysis, we exclude memory-mapped I/O since it is generally negligible.

2.6.2 File Accesses

One basic characteristic of our workloads is the division between reading and writing on open file descriptors. If an application uses an open file only for reading (or only for writing) or performs more activity on file descriptors of a certain type, then the file system may be able to utilize memory and allocate disk space in a more intelligent fashion.

To determine these characteristics, we classify each opened file descriptor based on the types of accesses (*i.e.*, read, write, or both) performed during its lifetime. We ignore the actual flags used when opening the file since we found they do not accurately reflect behavior; in all workloads, almost all write-only file descriptors were opened with `O_RDWR`. We measure both the proportional usage of each type of file descriptor and the relative amount of I/O performed on each.

Figure 2.9 shows how many file descriptors are used for each type of access. The overwhelming majority of file descriptors are used exclusively for reading or writing; read-write file descriptors are quite uncommon. Overall, read-only file descriptors are the most common across a majority of workloads; write-only file descriptors are popular in some iLife tasks, but are rarely used in iWork.

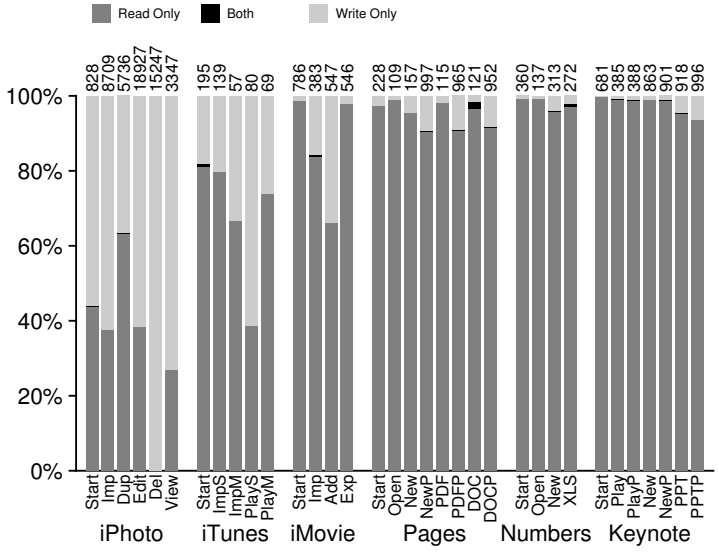

Figure 2.9: **Read/Write Distribution By File Descriptor.** *File descriptors can be used only for reads, only for writes, or for both operations. This plot shows the percentage of file descriptors in each category. This is based on usage, not* `open` *flags. Any duplicate file descriptors (e.g., created by* `dup`*) are treated as one and file descriptors on which the program does not perform any subsequent action are ignored.*

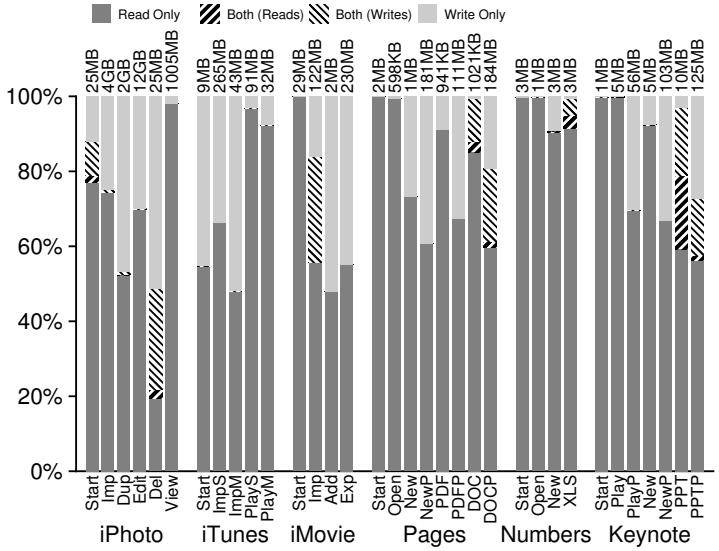

Figure 2.10: **Read/Write Distribution By Bytes.** *The graph shows how I/O bytes are distributed among the three access categories. The unshaded dark gray indicates bytes read as a part of read-only accesses. Similarly, unshaded light gray indicates bytes written in write-only accesses. The shaded regions represent bytes touched in read-write accesses, and are divided between bytes read and bytes written.*

We observe different patterns when analyzing the amount of I/O performed on each type of file descriptor, as shown in Figure 2.10. Although iWork tasks have very few write-only file descriptors, significant write I/O is often performed on those descriptors. Even though read-write file descriptors are rare, when present, they account for relatively large portions of total I/O (especially for exports to .doc, .xls, and .ppt). The read-write file descriptors are generally used for writing more than reading.

Conclusion: While many files are opened with the `O_RDWR` flag, most of them are subsequently accessed write-only; thus, file open flags cannot be used to predict how a file will be accessed. However, when an open file is both read and written by a task, the amount of traffic to that file occupies a significant portion of the total I/O. Finally, the rarity of read-write file descriptors may derive in part from the tendency of applications to write to a temporary file which they then rename as the target file, instead of overwriting the target file; we explore this tendency more in Section 2.8.2.

2.6.3 Read and Write Sizes

Another metric that affects file-system performance is the number and size of individual read and write system calls. If applications perform many small reads and writes, prefetching and caching may be a more effective strategy than it would be if applications tend to read or write entire files in one or two system calls. In addition, these data can augment some of our other findings, especially those regarding file sizes and complex access patterns.

We examine this by recording the number of bytes accessed by each read and write system call. As in our file size analysis in Section 2.5.2, we categorize these sizes into five groups: very small ($\leqslant$4 KB), small ($\leqslant$32 KB), medium ($\leqslant$128 KB), large ($\leqslant$1 MB), and very large ($>$1 MB).

Figures 2.11 and 2.12 display the number of reads and writes, respectively, of each size. We see very small operations dominating the iWork

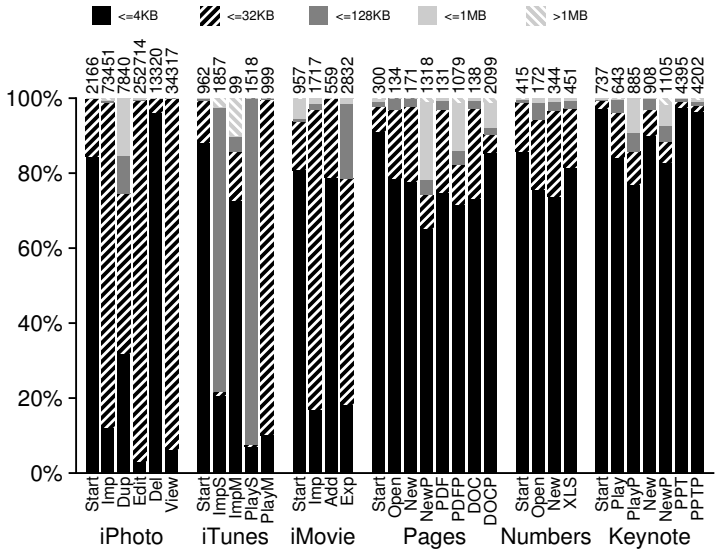

Figure 2.11: **Read Size Distribution.** *This plot groups individual read operations by the number of bytes each operation returns. The numbers atop the bars indicate the total number of read system calls in each task.*

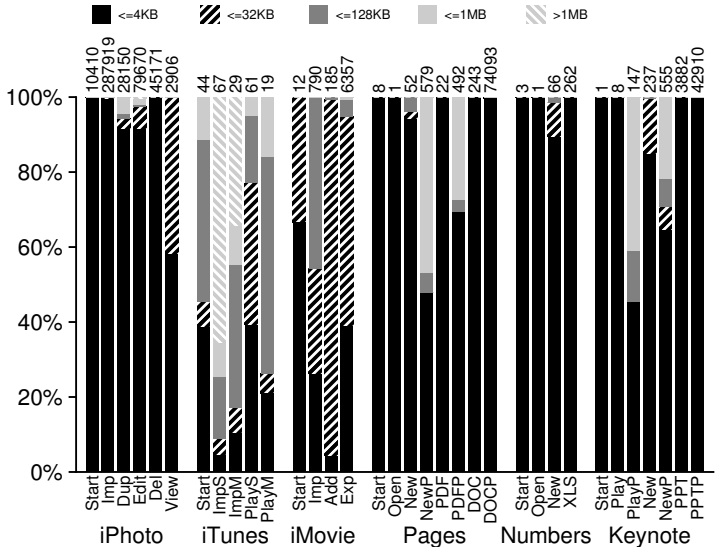

Figure 2.12: **Write Size Distribution.** *This plot groups individual write operations by the number of bytes each operation returns. The numbers atop the bars indicate the number of write system calls in each task.*

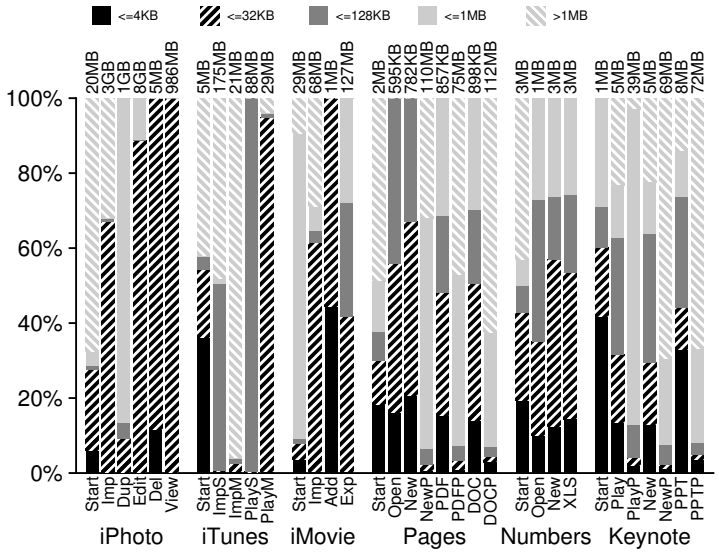

Figure 2.13: **Read Size Distribution by Bytes.** *This plot groups individual read operations by the percent of total bytes read for which each call is responsible. The numbers atop the bars indicate the total number of bytes read.*

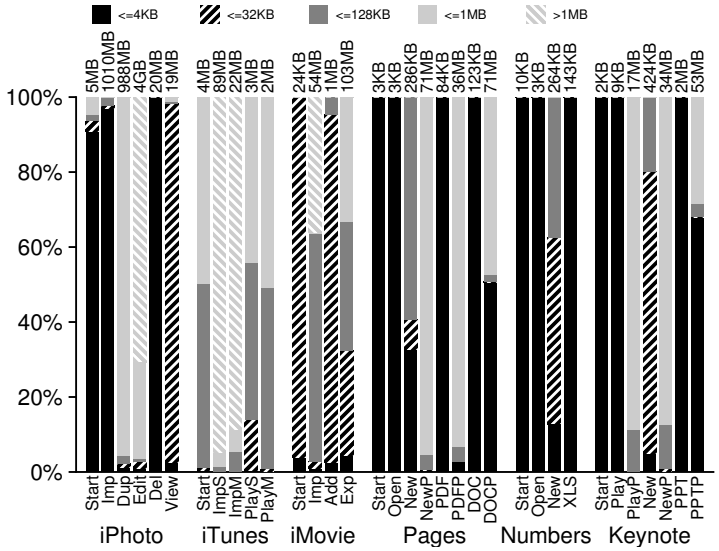

Figure 2.14: **Write Size Distribution by Bytes.** *This plot groups individual write operations by the percent of total bytes written each for which each call is responsible. The numbers atop the bars indicate the total number of bytes written.*

suite, with a more diverse variety of sizes in iLife; in particular, very large writes dominate iTunes ImpS because copying songs represents most the I/O performed during that task. Conversely, many iPhoto write tasks are composed almost entirely of very small operations, and reads for iLife tasks are usually dominated by a mixture of small and very small operations.

Figures 2.13 and 2.14 show the proportion of accessed bytes for each group. As with file sizes, large and very large reads cover substantial proportions of bytes in most of the workloads across both iLife and iWork. However, while large and very large writes account for sizable proportions of the workloads in iLife and the iWork workloads that use images, the vast majority of bytes in many of the other iWork tasks result from very small writes. Much of this is likely due to the complex patterns we observe when applications write to the complex files frequently used in productivity applications, although some of it also derives from the small amount of bytes that these tasks write.

Conclusion: We find that applications perform large numbers of very small ($\leqslant$4 KB) reads and writes; this fact is related to our finding in Section 2.5.2 that applications open many small files. While we also find that applications read most of their bytes in large requests, very small writes dominate many of the iWork workloads, due to a combination of the complex access patterns we have observed and the limited write I/O they perform.

2.6.4 Sequentiality

Historically, files have usually been read or written entirely sequentially [10]. We now determine whether sequential accesses are dominant in iBench. We measure this by examining all reads and writes performed on each file descriptor and noting the percentage of files accessed in strict sequential order (weighted by bytes).

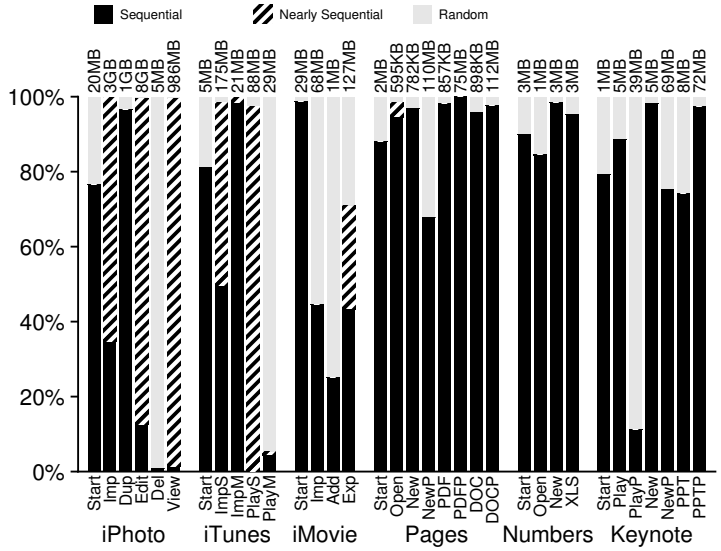

Figure 2.15: **Read Sequentiality.** *This plot shows the portion of file read accesses (weighted by bytes) that are sequentially accessed.*

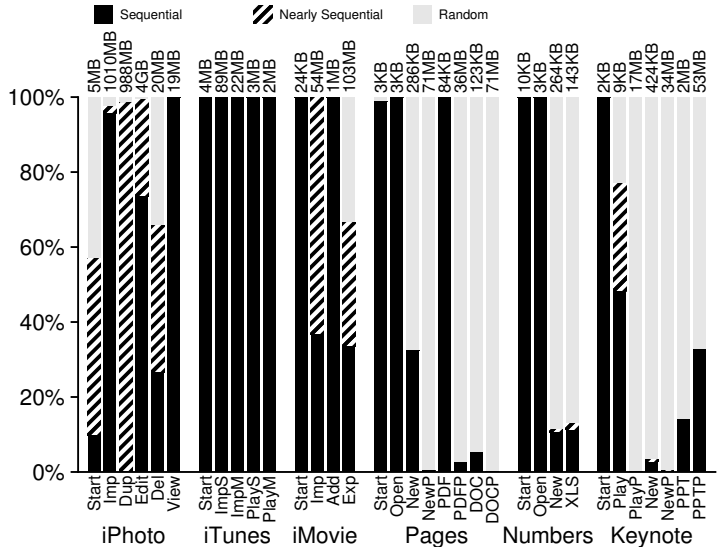

Figure 2.16: **Write Sequentiality.** *This plot shows the portion of file write accesses (weighted by bytes) that are sequentially accessed.*

We display our measurements for read and write sequentiality in Figures 2.15 and 2.16, respectively. The portions of the bars in black indicate the percent of file accesses that exhibit pure sequentiality. We observe high read sequentiality in iWork, but little in iLife (with the exception of the Start tasks and iTunes Import). The inverse is true for writes: while a majority of iLife writes are sequential, iWork writes are seldom sequential outside of Start tasks.

Investigating the access patterns to multimedia files more closely, we note that many iLife applications first touch a small header before accessing the entire file sequentially. To better reflect this behavior, we define an access to a file as "nearly sequential" when at least 95% of the bytes read or written to a file form a sequential run. We found that a large number of accesses fall into the "nearly sequential" category given a 95% threshold; the results do not change much with lower thresholds.

The slashed portions of the bars in Figures 2.15 and 2.16 show observed sequentiality with a 95% threshold. Tasks with heavy use of multimedia files exhibit greater sequentiality with the 95% threshold for both reading and writing. In several workloads (mainly iPhoto and iTunes), the I/O classified almost entirely as non-sequential with a 100% threshold is classified as nearly sequential. The difference for iWork applications is much less striking, indicating that accesses are more random.

In addition to this analysis of sequential and random accesses, we also measure how often a completely sequential access reads or writes an entire file. Figure 2.17 divides sequential reads into those that read the full file and those that only read part of it. In nearly all cases, the access reads the entire file; the only tasks for which sequential accesses of part of the file account for more than five percent of the total are iTunes Start and iMovie Imp and Exp. We omit the corresponding plot for writes, since virtually all sequential writes cover the entire file.

Conclusion: A substantial number of tasks contain purely sequential

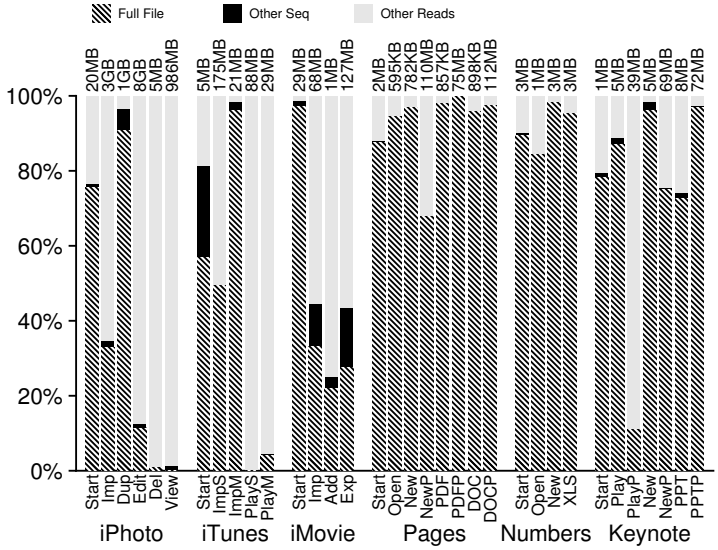

Figure 2.17: **Full-File Sequential Reads.** *This plot divides those reads that are fully sequential into partial reads of the file and reads of the entire file. The number atop each bar shows the total bytes the task read.*

accesses. When the definition of a sequential access is loosened such that only 95% of bytes must be consecutive, even more tasks contain primarily sequential accesses. These "nearly sequential" accesses result from metadata stored at the beginning of complex multimedia files: tasks frequently touch bytes near the beginning of multimedia files before sequentially reading or writing the bulk of the file. Those accesses that are fully sequential tend to access the entire file at once; applications that perform a substantial number of sequential reads of parts of files, like iMovie, often deal with relatively large files that would be impractical to read in full.

2.6.5 Preallocation

One of the difficulties file systems face when allocating contiguous space for files is not knowing how much data will be written to those files. Ap-

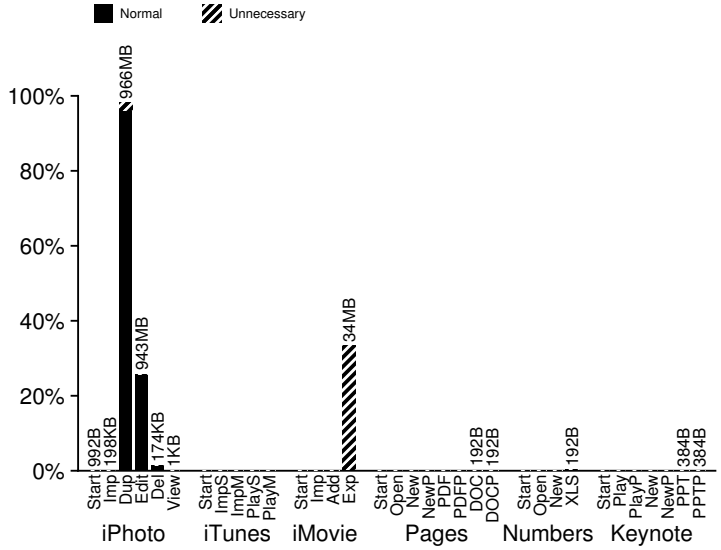

Figure 2.18: **Preallocation Hints.** *The sizes of the bars indicate which portion of file extensions are preallocations; unnecessary preallocations are diagonally striped. The number atop each bar indicates the absolute amount preallocated.*

plications can communicate this information by providing hints [80] to the file system to preallocate an appropriate amount of space. In this section, we quantify how often applications use preallocation hints and how often these hints are useful.

We instrument two calls usable for preallocation: `pwrite` and `ftruncate`. `pwrite` writes a single byte at an offset beyond the end of the file to indicate the future end of the file; `ftruncate` directly sets the file size. Sometimes a preallocation does not communicate anything useful to the file system because it is immediately followed by a single write call with all the data; we flag these preallocations as unnecessary.

Figure 2.18 shows the portion of file growth that is the result of preallocation. In all cases, preallocation was due to calls to `pwrite`; we never observed `ftruncate` preallocation. Overall, applications rarely preallocate space and the preallocations that occur are sometimes useless.

The three tasks with significant preallocation are iPhoto Dup, iPhoto Edit, and iMovie Exp. iPhoto Dup and Edit both call a `copyPath` function in the Cocoa library that preallocates a large amount of space and then copies data by reading and writing it in 1 MB chunks. iPhoto Dup sometimes uses `copyPath` to copy scaled-down images of size 50-100 KB; since these smaller files are copied with a single write, the preallocation does not communicate anything useful. iMovie Exp calls a QuickTime append function that preallocates space before writing the actual data; however, the data is appended in small 128 KB increments. Thus, the append is not split into multiple `write` calls; the preallocation is useless.

Conclusion: Although preallocation has the potential to be useful, few tasks use it to provide hints, and a significant number of the hints that are provided are useless. The hints are provided inconsistently: although iPhoto and iMovie both use preallocation for some tasks, neither application uses preallocation during import.

2.6.6 Open Durations

The average length of time that an application keeps its files open can be useful to know, as it provides a sense of how many resources the operating system should allocate to that file and its file descriptors. If applications keep file descriptors open for extended periods of time, closing a file descriptor likely means that the application is done with the file and that it no longer needs its contents; whereas if applications quickly close file descriptors after individual operations, the operation may not provide a meaningful hint for caching or buffering. Previous studies have found that files tend to be open for only brief periods of time [10], and we wish to determine whether this remains the case.

Due to the automated and compressed nature of our tasks, we cannot obtain a precise sense of how long file descriptors remain open under normal operation. We approximate this, however, by examining the

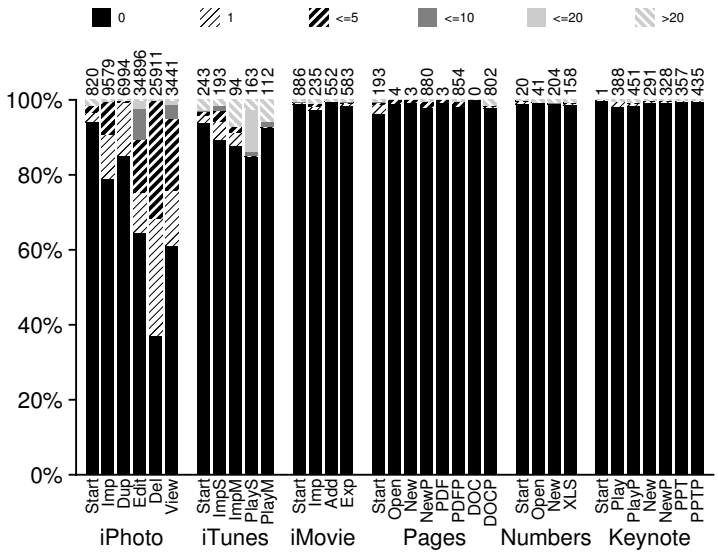

Figure 2.19: **Lifetime of Each File Open by Intervening Opens.** *This plot groups the file opens present in each task based on the number of opens to other files that occur during their lifetime. The number at the end of each bar displays the largest number of opens during the lifetime of any one open in each task.*

number of operations performed during the lifetime of each file descriptor. Specifically, we examine the number of intervening file opens during the lifetime of each access. As each task usually involves a large number of file descriptors, this should provide a reasonable sense of the relative lifetime of each open. We only track this statistic for file descriptors that access data, though the duration metric counts all intervening file opens.

We display our results in Figure 2.19. The vast majority of file descriptors in most tasks see no intervening file opens during their lifetimes. The only applications with tasks in which more than 5% of their file descriptors have others opened during their lifetimes are iPhoto and iTunes. Of these, iPhoto has significantly more file descriptors of this type, but most of them see fewer than five opens and many only one. iTunes, on the other hand, has fewer file descriptors with multiple intervening opens,

but those that do generally have significantly more intervening opens than those observed in iPhoto. This is particularly apparent in iTunes PlayS, where about 10% of the file descriptors see between 11 and 20 opens during their lifetimes.

Despite the predominance of file descriptors without intervening opens, all experiments except Pages Open, New, PDF, and DOC and Keynote Start feature at least one file open over the lifetime of a large number of other file opens. The nature of this file varies: in iPhoto and iWork, it is generally a database, while in iTunes it is a resource fork of an iTunes-specific file, and in iMovie, it is generally the movie being manipulated (except for Start, where it is a database).

The two applications that open the most files while others are open, iPhoto and iTunes, also make the most use of multi-threaded I/O, as shown in Section 2.9. This indicates that these applications may perform I/O in parallel. Similarly, the prevalence of file descriptors without intervening opens indicates that most I/O operations are likely performed serially.

Conclusion: In the vast majority of cases, applications close the file they are working on before opening another, though they frequently keep one or two files open for the majority of the task. Those applications that do open multiple file descriptors in succession tend to perform substantial multi-threaded I/O, indicating that the opens may represent parallel access to files instead of longer-lived accesses.

2.6.7 Metadata Access

Reading and writing file data hardly comprises the entirety of file system operations; many file system calls are devoted to manipulating file metadata. As previous studies have noted [55], `stat` calls (including variations like `fstat`, `stat64`, and `lstat`) often occupy large portions of file system workloads, as they provide an easy way to verify whether a file

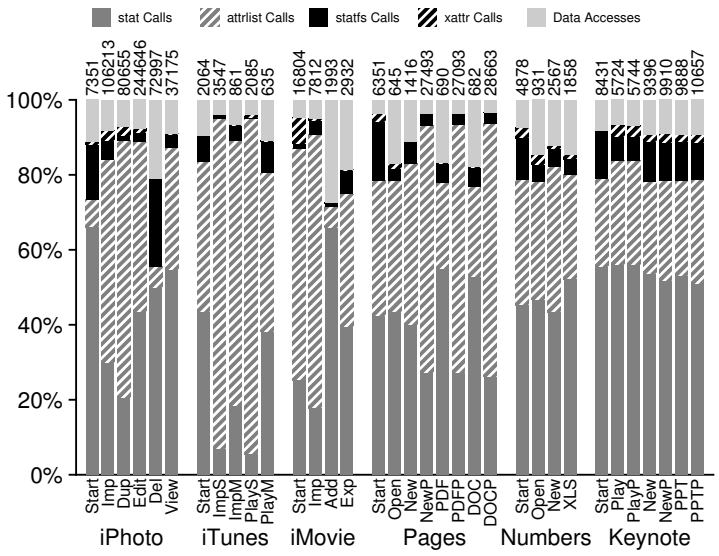

Figure 2.20: **Relative Quantities of Metadata Access to Data Access.** *This plot shows the ratios of* `stat` *calls,* `statfs` *calls, and* `xattr` *accesses to data accesses in each task. The total combined amount of these is provided at the end of each bar.*

currently exists and to examine the attributes of a file before opening it. Similarly, `statfs` offers an interface to access metadata about the file system on which a given file resides. Mac OS X provides an additional function, `getattrlist`, which combines the data returned in these two calls, as well as providing additional information; some of the attributes that it deals with can be manipulated by the application with the `setattrlist` call. Finally, the `xattr` related calls allow applications to retrieve and set arbitrary key-value metadata on each file. To determine how heavily applications use these metadata operations, we compare the relative frequency of these calls to the frequency of file data accesses.

We display our results in Figure 2.20. Metadata accesses occur substantially more often than data accesses (*i.e.*, open file descriptors which receive at least one read or write request) with calls to `stat` and `getattrlist`

together comprising close to 80% of the access types for most workloads. In a majority of the iLife workloads, calls to `getattrlist` substantially exceed those to `stat`; however, `stat` calls hold a plurality in most iWork workloads, except the Pages tasks that deal with images. Data accesses are the third-most common category and generally occupy 10-20% of all accesses, peaking at 30% in iTunes PlayS. `statfs` calls, while uncommon, appear noticeably in all workloads except iTunes PlayS; at their largest, in iPhoto Del, they occupy close to 25% of the categories. Finally, `xattr` calls are the rarest, seldom comprising more than 3-4% of accesses in a workload, though still appearing in all tasks other than those in iTunes (in several cases, such as iPhoto Del, they comprise such a small portion of the overall workload that they do not appear in the plot).

Conclusion: As seen in previous studies, metadata accesses are very common, greatly outnumbering accesses to file data across all of our workloads. `stat` is no longer as predominant as it has been, however; in many cases, `attrlist` calls appear more frequently. `statfs` and `xattr` access are not nearly as common, but still appear in almost all of the workloads. As there are usually at least five metadata accesses for each data access. The need to keep these optimize these system calls, as described by Jacob *et al.* [55], remains.

2.7 Memory: Caching, Buffering, and Prefetching

In this section, we explore application behavior that affects file-system memory management. We measure how often previously accessed data is read again in the future, how often applications overwrite previously written data, and how frequently hints are provided about future patterns. These measures have implications for caching, write buffering, and prefetching strategies.

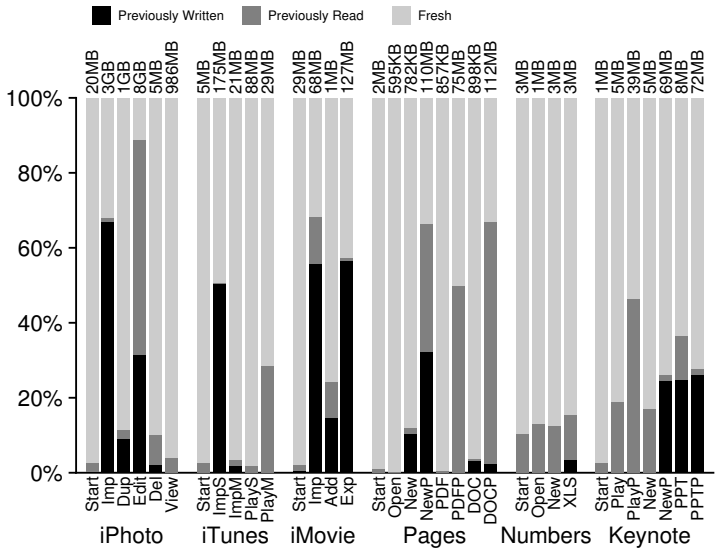

Figure 2.21: **Reads of Previously Accessed Data.** *This plots shows what portion of reads are to file regions that were previously read or written. The numbers atop the bars represent all the bytes read by the tasks. The bars are broken down to show how much of this read data was previously read or written. Data previously read and written is simply counted as previously written.*

2.7.1 Reuse and Overwrites

File systems use memory to cache data that may be read in the future and to buffer writes that have not yet been flushed to disk. Choosing which data to cache or buffer will affect performance; in particular, we want to cache data that will be reused, and we want to buffer writes that will soon be invalidated by overwrites. In this section, we measure how often tasks reuse and overwrite data. We keep track of which regions of files are accessed at byte granularity.

Figure 2.21 shows what portion of all reads are to data that was previously accessed. For most tasks, 75-100% of reads are to fresh data, but for eight tasks, about half or more of the data was previously accessed. The previously accessed data across all tasks can be divided somewhat

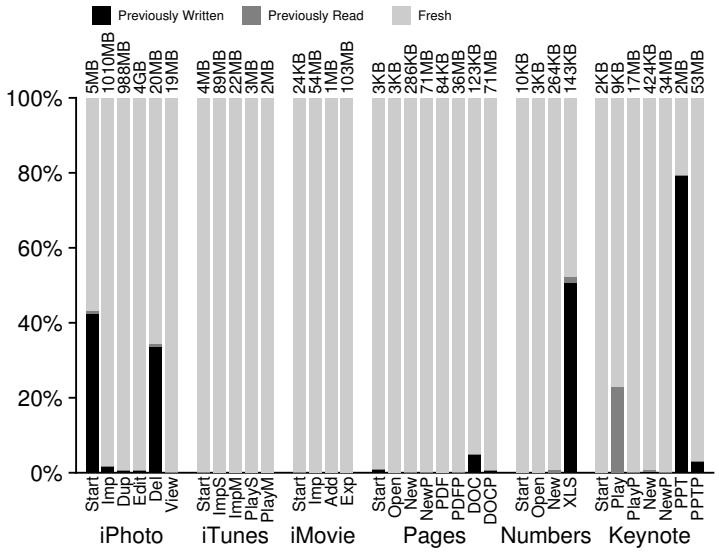

Figure 2.22: **Overwrites of Previously Accessed Data.** *This plots shows what portion of writes are to file regions that were previously read or written. The numbers atop the bars represent all the bytes written by the tasks. The bars are broken down to show how many of these writes overwrite data that was previously read or written. Data previously read and written is simply counted as previously written.*

evenly into *previously read* and *previously written* categories.

Several of the tasks that exhibit the read-after-write pattern are importing data to an application library (*i.e.*, iPhoto Imp, iTunes ImpS, and iMovie Imp). For example, iPhoto Import first performs reads and writes to copy photos to a designated library directory. The freshly created library copies are then read in order to generate thumbnails. It would be more efficient to read the original images once and create the library copies and thumbnails simultaneously. Unfortunately, use of two independent high-level abstractions, *copy-to-library* and *generate-thumbnail*, cause additional I/O.

Figure 2.22 shows what portion of all writes are overwriting data that

has previously been read or written. We see that such overwrites are generally very rare, with four exceptions: iPhoto Start, iPhoto Delete, Numbers XLS, and Keynote PPT.

For the iPhoto tasks, the overwrites are caused by SQLite. For Numbers XLS and Keynote PPT, the overwrites are caused by an undocumented library, `SFCompatability`, which is used to export to Microsoft formats. When only a few bytes are written to a file using the `streamWrite` function of this library, the function reads a 512-byte page, updates it, and writes it back. We noted this behavior for Pages DOCP in the case study in Section 2.3; presumably the 512-byte granularity for the read-update-write operation is due to the Microsoft file format, which resembles a FAT file system with 512-byte pages. The read-update-write behavior occurs in all the tasks that export to Microsoft formats; however, the repetitious writes are less frequent for Pages. Also, when images are in the exported file, writes to the image sections dwarf the small repetitious writes. Thus, the repetitious write behavior caused by `SFCompatability` is only pronounced for Numbers XLS and Keynote PPT.

Conclusion: A moderate amount of reads could potentially be serviced by a cache, but most reads are to fresh data, so techniques, such as intelligent disk allocation, are necessary to guarantee quick access to uncached data. Written data is rarely overwritten, so waiting to flush buffers until data becomes irrelevant is usually not helpful. These findings also have implications for SSD design: overwrites will not be a frequent cause of page invalidation. Finally, we observe, many of the reads and writes to previously accessed data which do occur are due to I/O libraries and high-level abstractions, and could be eliminated by combining I/O tasks.

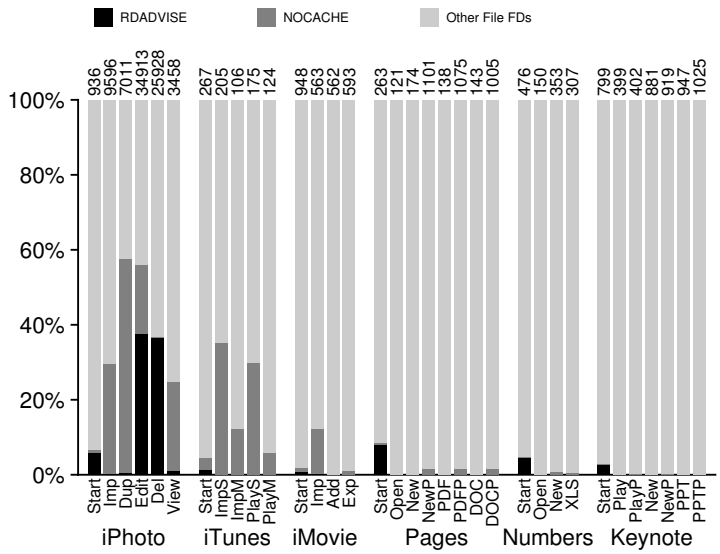

Figure 2.23: **File Descriptors Given Caching Commands.** *This plot shows the percent of file descriptors issued* `fcntl` *system calls with the* `F_RDADVISE` *and* `F_NOCACHE` *commands, distinguishing between file descriptors that eventually have F_NOCACHE disabled and those that only have it enabled. The numbers atop the bars indicate the total number of file descriptors opened on files.*

2.7.2 Caching Hints

Accurate caching and prefetching behavior can significantly affect the performance of a file system, improving access times dramatically for files that are accessed repeatedly. Conversely, if a file will only be accessed once, caching data wastes memory that could be better allocated. Correctly determining the appropriate behavior can be difficult for the file system without domain-specific knowledge. Thus, Mac OS X allows developers to affect the caching behavior of the file system through two commands associated with the `fcntl` system call, `F_NOCACHE` and `F_RDADVISE`. `F_NOCACHE` allows developers to explicitly disable and enable caching for certain file descriptors, which is useful if the developer knows that either

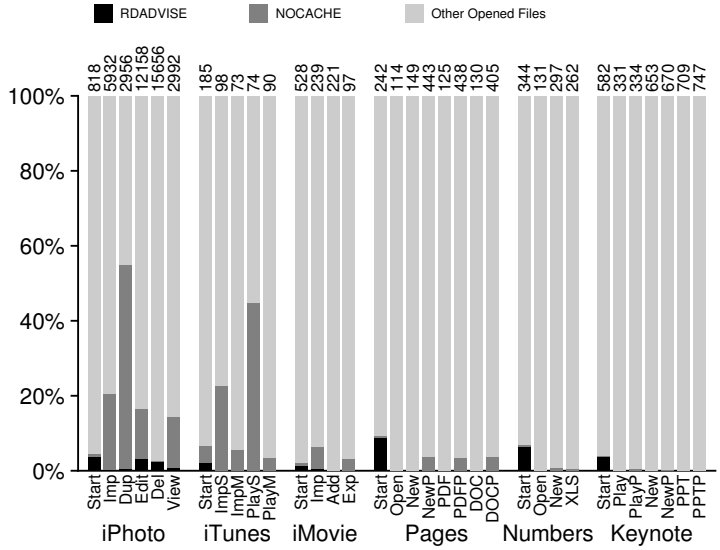

Figure 2.24: **Inodes Affected By Caching Commands.** *This plot shows the percent of the inodes each task touches that are opened by file descriptors with either* `F_RDADVISE` *or* `F_NOCACHE`*. The numbers atop the bars indicate the total inodes touched by each task.*

all or a portion of the file will not be reread. `F_RDADVISE` suggests an asynchronous read to prefetch data from the file into the page cache. These commands are only helpful, however, if developers make active use of them, so we analyze the frequency with which they appear in our traces.

Figure 2.23 displays the percent of file descriptors with `F_RDADVISE` issued, `F_NOCACHE` enabled, and `F_NOCACHE` both enabled and disabled during their lifetimes. The figure also includes opened file descriptors which received no I/O, even though most of our plots exclude them (sometimes files are opened just so an `F_RDADVISE` can be issued). We observed no file descriptors where `F_NOCACHE` was combined with `F_RDADVISE`. Overall, we see these commands used most heavily in iPhoto and iTunes. In particular, over half of the file descriptors opened on files in iPhoto Dup and Edit receive one of these commands, with `F_NOCACHE` overwhelmingly

dominating Dup and F_RDADVISE dominating Edit. Most of the other iLife tasks tend to use F_NOCACHE much more frequently than F_RDADVISE, with the exception of iPhoto Start and Del. In contrast, only the Start workloads of the iLife applications issue any of these commands to more than one or two percent of their file descriptors, with F_RDADVISE occurring most frequently. When F_NOCACHE occurs (usually in those applications dealing with photos), it is usually disabled before the file descriptor is closed.

To complement this, we also examine the relative number of inodes in each workload that receive these commands. Figure 2.24 shows our results. F_RDADVISE is generally issued to a much smaller proportion of the total inodes than total file descriptors, indicating that advisory reads are repeatedly issued to a small set of inodes in the workloads in which they appear. In contrast, the proportion of inodes affected by F_NOCACHE is generally comparable to (or, in the case of iTunes PlayS, greater than) the proportion of file descriptors, which shows that these inodes are usually only opened once.

Conclusion: Mac OS X allows developers to guide the file system's caching behavior using the fcntl commands F_NOCACHE and F_RDADVISE. Only iPhoto and iTunes make significant use of them, though all of the iBench applications use them in at least some of their tasks. When the commands are used, they generally occur in ways that make sense: files with caching disabled tend not to be opened more than once, whereas files that receive advisory reads are repeatedly opened. Thus, developers are able to make effective use of these primitives when they choose to do so.

2.8 Transactional Properties

In this section, we explore the degree to which the iBench tasks require transactional properties from the underlying file and storage system. In

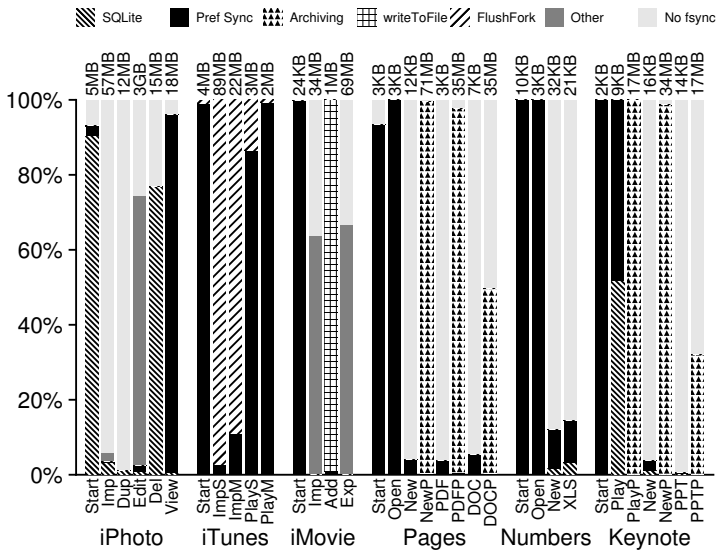

Figure 2.25: **Percentage of Fsync Bytes.** *The percentage of `fsync`'d bytes written to file descriptors is shown, broken down by cause. The value atop each bar shows total bytes synchronized.*

particular, we investigate the extent to which applications require writes to be durable; that is, how frequently they invoke calls to `fsync` and which APIs perform these calls. We also investigate the atomicity requirements of the applications, whether from renaming files or exchanging inodes. Finally, we explore how applications use file locking to achieve isolation.

2.8.1 Durability

Writes typically involve a trade-off between performance and durability. Applications that require write operations to complete quickly can write data to the file system's main memory buffers, which are lazily copied to the underlying storage system at a convenient time. Buffering writes in main memory has a wide range of performance advantages: writes to the same block may be coalesced, writes to files that are later deleted need

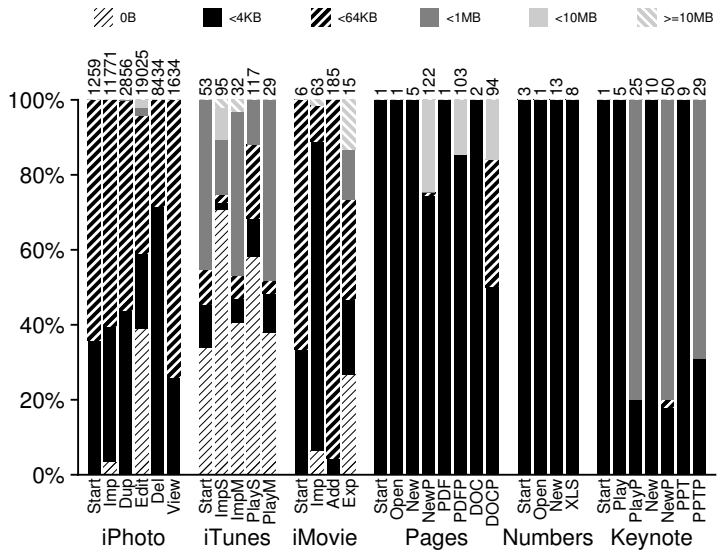

Figure 2.26: **Fsync Sizes.** *This plot shows a distribution of* `fsync` *sizes. The total number of* `fsync` *calls appears at the end of the bars.*

not be performed, and random writes can be more efficiently scheduled.

On the other hand, applications that rely on durable writes can flush written data to the underlying storage layer with the `fsync` system call. The frequency of `fsync` calls and the number of bytes they synchronize directly affect performance: if `fsync` appears often and flushes only several bytes, then performance will suffer. Therefore, we investigate how modern applications use `fsync`.

Figure 2.25 shows the percentage of written data each task synchronizes with `fsync`. The graph further subdivides the source of the `fsync` activity into six categories. *SQLite* indicates that the SQLite database engine is responsible for calling `fsync`; *Archiving* indicates an archiving library frequently used when accessing ZIP formats; *Pref Sync* is the `PreferencesSynchronize` function call from the Cocoa library; *writeToFile* is the Cocoa call `writeToFile` with the `atomically` flag set; and finally, *FlushFork* is the Carbon `FSFlushFork` routine.

At the highest level, the figure indicates that half the tasks synchronize close to 100% of their written data while approximately two-thirds synchronize more than 60%. iLife tasks tend to synchronize many megabytes of data, while iWork tasks usually only synchronize tens of kilobytes (excluding tasks that handle images).

To delve into the APIs responsible for the `fsync` calls, we examine how each bar is subdivided. In iLife, the sources of `fsync` calls are quite varied: every category of API except for Archiving is represented in one of the tasks, and many of the tasks call multiple APIs which invoke `fsync`. In iWork, the sources are more consistent; the only sources are Pref Sync, SQLite, and Archiving (for manipulating compressed data).

Given that these tasks require durability for a significant percentage of their write traffic, we next investigate the frequency of `fsync` calls and how much data each individual call pushes to disk. Figure 2.26 groups `fsync` calls based on the amount of I/O performed on each file descriptor when `fsync` is called, and displays the relative percentage each category comprises of the total I/O.

These results show that iLife tasks call `fsync` frequently (from tens to thousands of times), while iWork tasks call `fsync` infrequently except when dealing with images. From these observations, we infer that calls to `fsync` are mostly associated with media. The majority of calls to `fsync` synchronize small amounts of data; only a few iLife tasks synchronize more than a megabyte of data in a single `fsync` call.

Conclusion: All the applications we study aggressively flush data to disk with `fsync`. This behavior is especially problematic for file systems because the amount of data flushed per `fsync` call is often quite small. Based on our analysis of the source of `fsync` calls, many calls may be incidental and an unintentional side-effect of the API (*e.g.*, those from SQLite or Pref Sync), but many are performed intentionally by the programmer. Furthermore, some of the tasks synchronize small amounts of data fre-

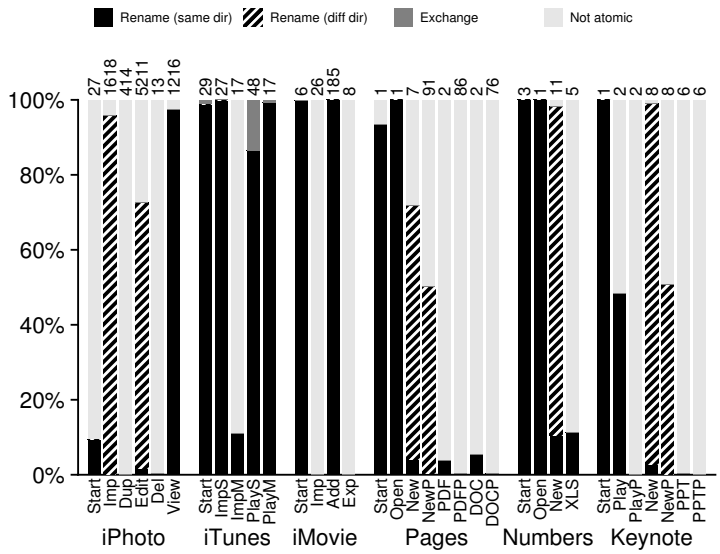

Figure 2.27: **Atomic Writes.** *The portion of written bytes written atomically is shown, divided into groups: (1)* `rename` *leaving a file in the same directory; (2)* `rename` *causing a file to change directories; (3)* `exchangedata`*, which never causes a directory change. The atomic file-write count appears atop each bar.*

quently, presenting a challenge for file systems.

2.8.2 Atomic Writes

Applications often require file changes to be atomic. In this section, we quantify how frequently applications use different techniques to achieve atomicity. We also identify cases where performing writes atomically can interfere with directory locality optimizations by moving files from their original directories. Finally, we identify the causes of atomic writes.

Applications can atomically update a file by first writing the desired contents to a temporary file and then using either the `rename` or `exchangedata` call to atomically replace the old file with the new file. With `rename`, the new file is given the same name as the old, deleting the original and

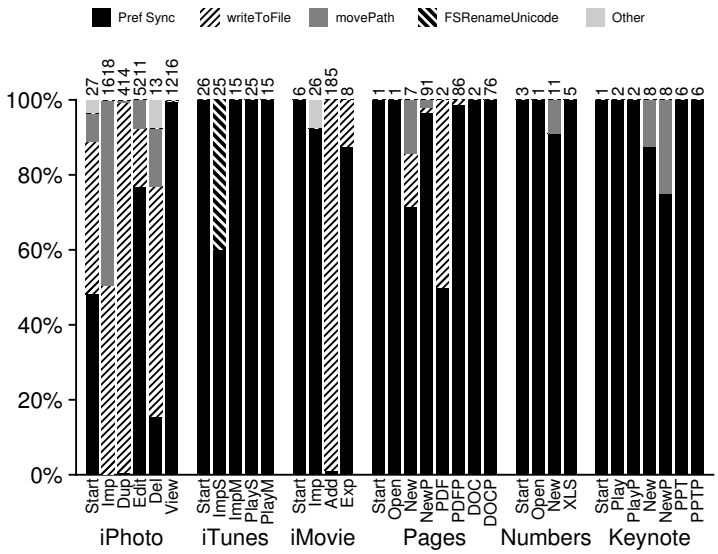

Figure 2.28: **Rename Causes.** *This plot shows the portion of* `rename` *calls caused by each of the top four higher level functions used for atomic writes. The number of* `rename` *calls appears at the end of the bars.*

replacing it. With `exchangedata`, the inode numbers assigned to the old file and the temporary file are swapped, causing the old path to point to the new data; this allows the file path to remain associated with the original inode number, which is necessary for some applications.

Figure 2.27 shows how much write I/O is performed atomically with `rename` or `exchangedata`; `rename` calls are further subdivided into those which keep the file in the same directory and those which do not. The results show that atomic writes are quite popular and that, in many workloads, all the writes are atomic. The breakdown of each bar shows that `rename` is frequent; many of these calls move files between directories. `exchangedata` is rare and used only by iTunes for a small fraction of file updates.

We find that most of the `rename` calls causing directory changes occur when a file (*e.g.*, a document or spreadsheet) is saved at the user's request.

We suspect different directories are used so that users are not confused by seeing temporary files in their personal directories. Interestingly, atomic writes are performed when saving to Apple formats, but not when exporting to Microsoft formats. We suspect the interface between applications and the Microsoft libraries does not specify atomic operations well.

Figure 2.28 identifies the APIs responsible for atomic writes via `rename`. *Pref Sync*, from the Cocoa library, allows applications to save user and system wide settings in .plist files. *WriteToFile* and *movePath* are Cocoa routines and *FSRenameUnicode* is a Carbon routine. A solid majority of the atomic writes are caused by Pref Sync; this is an example of I/O behavior caused by the API rather than explicit programmer intention. The second most common atomic writer is writeToFile; in this case, the programmer is requesting atomicity but leaving the technique up to the library. Finally, in a small minority of cases, programmers perform atomic writes themselves by calling movePath or FSRenameUnicode, both of which are essentially `rename` wrappers.

Conclusion: Many of our tasks write data atomically, generally doing so by calling `rename`. The bulk of atomic writes result from API calls; while some of these hide the underlying nature of the write, others make it clear that they act atomically. Thus, developers desire atomicity for many operations, and file systems will need to either address the ensuing renames that accompany it or provide an alternative mechanism for writing atomically. In addition, the absence of atomic writes when writing to Microsoft formats highlights the inconsistencies that can result from the use of high level libraries.

2.8.3 Isolation via File Locks

Concurrent I/O to the same file by multiple processes can yield unexpected results. For correctness, we need isolation between processes. Towards this end, UNIX file systems provide an advisory-locking API, which

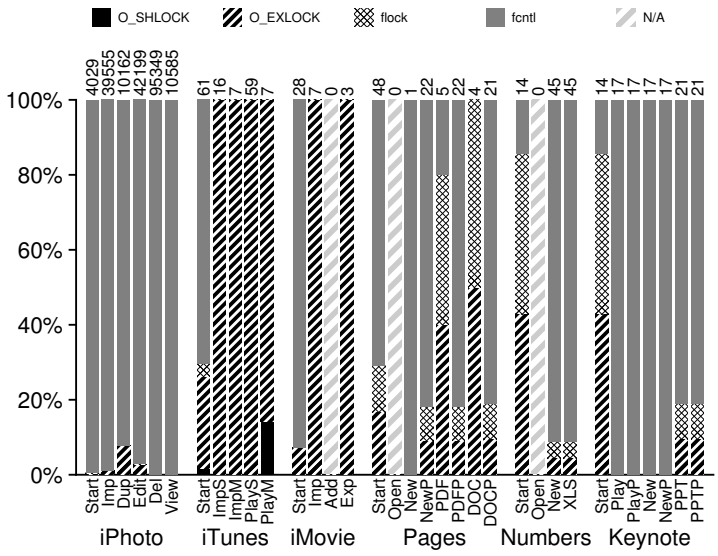

Figure 2.29: **Locking Operations.** *The explicit calls to the locking API are shown, broken down by type. O_SHLOCK and O_EXLOCK represent calls to* `open` *with those flags, flock represents a change to a file's lock status via a call to* `flock`*, and fcntl represents file-region locking via* `fcntl` *with certain commands (*`F_GETLK`*,* `F_SETLK`*, or* `F_SETLKW`*). The number atop each bar indicates the number of locking calls.*

achieves mutual exclusion between processes that use the API. However, because the API is advisory, its use is optional, and processes are free to ignore its locks. The API supports both whole-file locking and file-region locking. File-region locking does not inherently correspond to byte regions in the file; instead, applications are free define their own semantics for the regions locked (*e.g.*, a lock of size ten could cover ten records in the file, each of which is 100 bytes long). We explore how the iBench applications use these locking API calls.

Figure 2.29 shows the frequency and type of explicit locking operations (implicit unlocks are performed when file descriptors are closed, but we do not count these). Most tasks perform 15-50 lock or unlock opera-

tions; only three tasks do not use file locks at all. iPhoto makes extreme use of locks; except for the Start task, all the iPhoto tasks make tens of thousands of calls through the locking API.

We observe that most calls are issued via the `fcntl` system call; these calls lock file regions. Whole-file locking is also used occasionally via the `O_SHLOCK` and `O_EXLOCK` `open` flags; the vast majority of these whole-file lock operations are exclusive. `flock` can be used to change the lock status on a file after it has been opened; these calls are less frequent, and they are only used to unlock a file that was already locked by a flag passed to `open`.

The extreme use of file-region locks by iPhoto is due to iPhoto's dependence on SQLite. Many database engines are server based; in such systems, the server can provide isolation by doing its own locking. In contrast, SQLite has no server, and multiple processes may concurrently access the same database files directly. Thus, file-system locking is a necessary part of the SQLite design [101].

Conclusion: Most tasks make some use of the locking API, and file-region locking accounts for the majority of this use. Tasks that heavily rely on SQLite involve numerous lock operations. In the case of iPhoto, it seems unlikely that other applications need to access iPhoto's database files. iPhoto would be more efficient if there were a single global lock that prevented multiple instances of the application from running concurrently so that SQLite could be configured to skip file locking. iPhoto is a prime example of how modularity can result in excessive fine-grained locking within subcomponents.

2.9 Threads and Asynchronicity

Home-user applications are interactive and need to avoid blocking when I/O is performed. Asynchronous I/O and threads are often used to hide

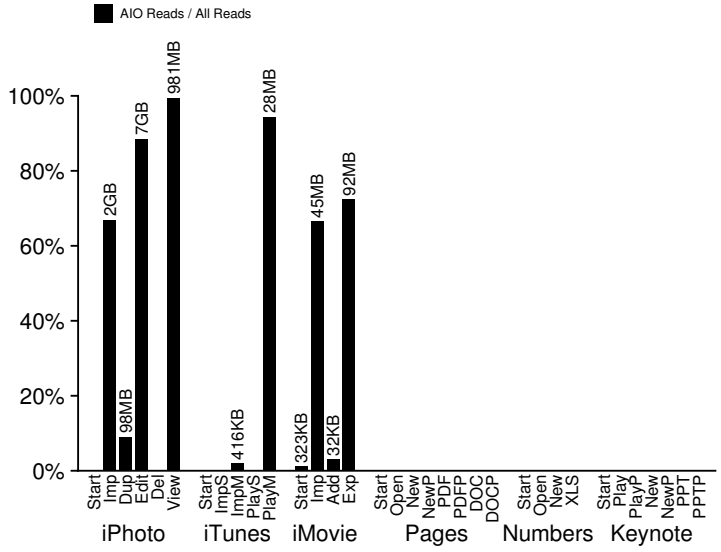

Figure 2.30: **Asynchronous Reads.** *This plot shows the percentage of read bytes read asynchronously via* `aio_read`*. The total amount of asynchronous I/O is provided at the end of the bars.*

the latency of slow operations from users. For our final experiments, we investigate how often applications use asynchronous I/O libraries or multiple threads to avoid blocking.

Figure 2.30 shows the relative amount of read operations performed asynchronously with `aio_read`; none of the tasks use `aio_write`. We find that asynchronous I/O is used rarely and only by iLife applications. However, in those cases where asynchronous I/O is performed, it is used quite heavily.

Figure 2.31 investigates how threads are used by these tasks: specifically, the portion of I/O performed by each of the threads. The numbers at the tops of the bars report the number of threads performing I/O. iPhoto and iTunes leverage a significant number of threads for I/O, since many of their tasks are readily subdivided (*e.g.*, importing 400 different photos). Only a handful of tasks perform all their I/O from a single thread. For

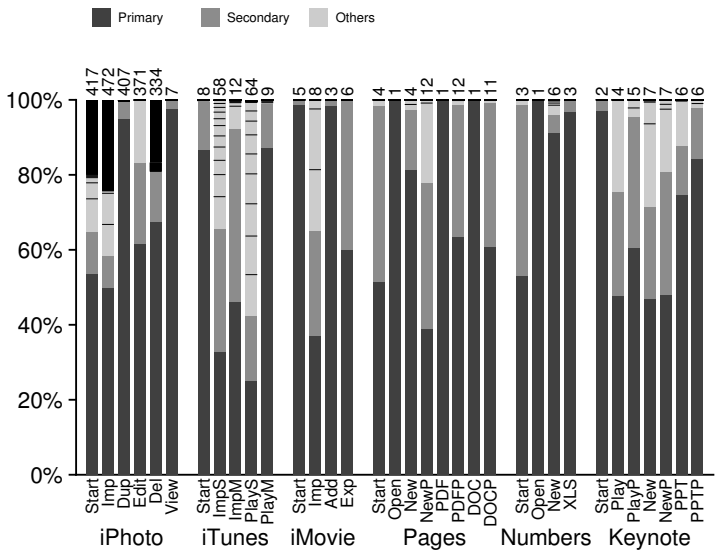

Figure 2.31: **I/O Distribution Among Threads.** *The stacked bars indicate the percentage of total I/O performed by each thread. The I/O from the threads that do the most and second most I/O are dark and medium gray respectively, and the other threads are light gray. Black lines divide the I/O across the latter group; black areas appear when numerous threads do small amounts of I/O. The total number of threads that perform I/O is indicated next to the bars.*

most tasks, a small number of threads are responsible for the majority of I/O.

Figure 2.32 shows the responsibilities of each thread that performs I/O, where a thread can be responsible for reading, writing, or both. Significantly more threads are devoted to reading than to writing, with a fair number of threads responsible for both. This indicates that threads are the preferred technique to avoiding blocking and that applications may be particularly concerned with avoiding blocking due to reads.

Conclusion: Our results indicate that iBench tasks are concerned with hiding long-latency operations from interactive users and that threads are the preferred method for doing so. Virtually all of the applications we

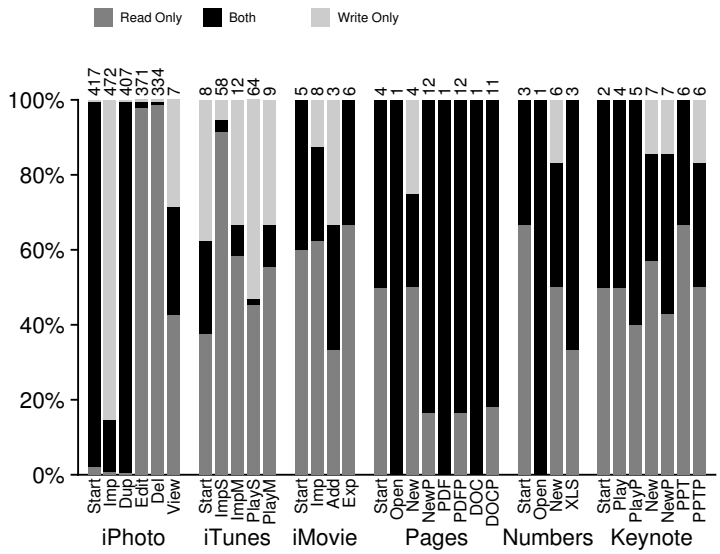

Figure 2.32: **Thread Type Distribution.** *The plot categorizes threads that do I/O into three groups: threads that read, threads that write, or threads that both read and write. The total number of threads that perform I/O is indicated next to the bars.*

study issue I/O requests from multiple threads, and some launch I/Os from hundreds of different threads.

2.10 Summary

We began our analysis by taking one task as a case study (Pages saving a .doc file) and considering its I/O patterns in great detail (§2.3). From this case study, we made seven observations. We summarize the rest of our analysis by commenting on the degree to which the other 33 tasks exhibit our seven findings.

A file is not a file. We saw that when Pages saves a .doc file, the saved file is formatted as a file-system image, but backed by a regular file. These kinds of files never represent a large number of the files accessed, but

complex files do receive a disproportionality large amount of I/O whenever one of the iWork applications is saving to a Microsoft format. iLife does not appear to use files that are formatted as file-system images. However, all six application we studied heavily used .plist files to store key-value pairs; greater efficiency could perhaps be achieved if file systems understood these patterns or otherwise provided special support for this use case.

Sequential access is not sequential. Multimedia files are a very important part of the I/O performed by all six applications, even when the task does not specifically involve manipulating multimedia, because all the applications provide a rich GUI. Accesses to multimedia files often involve random access to metadata near the beginning of files and large sequential access to the rest of the file. These accesses can easily be misclassified, so we contribute an alternative definition of "nearly sequentiality" that improves the understanding of accesses to multimedia files. This is important for the iLife applications, but very little I/O is nearly sequential for iWork.

Auxiliary files dominate. Almost all the tasks involve accessing hundreds of files, even though the purpose of a task usually involves very few files of which the user would actually be aware.

Writes are often forced. This is very generally true for both iLife and iWork; a majority of writes are forced to disk in a majority of the tasks. The use of `fsync` poses a challenge to systems. Often, file systems buffer updates and perform batch writes at regular intervals, but `fsync` renders this strategy ineffective.

Renaming is popular. This is also very generally true for all the applications. About half the tasks rename a majority of the written data in order to achieve atomicity. The use of `rename` for atomicity means that files are constantly being created and deleted with each minor update, potentially leading to external fragmentation and poor data placement.

Furthermore, atomic writes that involve moving a new file from a temporary directory to its final destination could easily trick naive systems into poor data placement.

Multiple threads perform I/O. In only four of the tasks we analyzed does the application issue all I/O from a single thread. However, the number of threads is usually not excessive: about two thirds of the tasks have fewer than ten threads performing I/O.

Frameworks influence I/O. We found that many of the demands for durability and atomicity placed on the file systems and various bizarre access patterns were the result of high level API use, not necessarily the informed intent of the programmer. For example, most atomic writes and many `fsync` requests resulted from use of a preferences library that allow programmers to store key-value pairs per user. It is unlikely that the programmer always needed the atomicity and durability this library automatically provided.

3
Facebook Messages Measurement

Storage systems are often organized as a composition of layers. In Chapter 2, we studied the impact of libraries on I/O. Many of those libraries were user-space layers that happen to run in the same address space as the application. In this chapter, we study the composition of storage layers that run as distinct subsystems in separate fault domains. Whereas the libraries of the previous chapter interact via regular function calls, the layers of this chapter communicate via well-defined RPC and system-call interfaces.

We focus our study on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from Bigtable [20]) atop a distributed file system (HDFS [97], derived from the Google File System [40]). Our goal is to study the interaction of these important systems, with a particular focus on the lower layer; thus, our highest-level question: *is HDFS an effective storage backend for HBase?*

To derive insight into this hierarchical system, and thus answer this question, we trace and analyze it under a popular workload: Facebook Messages (FM) [72]. FM is a messaging system that enables Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. FM stores its information within HBase (and thus, HDFS), and hence serves as an excellent case study.

To complement our analysis, we also perform numerous simulations of various caching, logging, and other architectural enhancements and modifications. Through simulation, we can explore a range of "what if?" scenarios, and thus gain deeper insight into the efficacy of the layered storage system.

The rest of this chapter is organized as follows. First, a background section describes HBase and the Messages storage architecture (§3.1). Then we describe our methodology for tracing, analysis, and simulation (§3.2). We present our analysis results (§3.3), make a case for adding a flash tier (§3.4), and measure layering costs (§3.5). Finally, we summarize our findings (§3.6).

3.1 Background

We now describe the HBase sparse-table abstraction (§3.1.1) and the overall FM storage architecture (§3.1.2).

3.1.1 Versioned Sparse Tables

HBase, like Bigtable [20], provides a *versioned sparse-table* interface, which is much like an associative array, but with two major differences: (1) keys are ordered, so lexicographically adjacent keys will be stored in the same area of physical storage, and (2) keys have semantic meaning which influences how HBase treats the data. Keys are of the form *row:column:version*. A *row* may be any byte string, while a *column* is of the form *family:name*. While both column families and names may be arbitrary strings, families are typically defined statically by a schema while new column names are often created during runtime. Together, a row and column specify a cell, for which there may be many versions.

A sparse table is sharded along both row and column dimensions. Rows are grouped into *regions*, which are responsible for all the rows

within a given row-key range. Data is sharded across different machines with region granularity. Regions may be split and re-assigned to machines with a utility or automatically upon reboots. Columns are grouped into families so that the application may specify different policies for each group (*e.g.*, what compression to use). Families also provide a locality hint: HBase clusters together data of the same family.

3.1.2 Messages Architecture

Users of FM interact with a web layer, which is backed by an application cluster, which in turn stores data in a separate HBase cluster. The application cluster executes FM-specific logic and caches HBase rows while HBase itself is responsible for persisting most data. Large objects (*e.g.*, message attachments) are an exception; these are stored in Haystack [12] because HBase is inefficient for large data (§3.3.1). This design applies Lampson's advice to "handle normal and worst case separately" [61].

HBase stores its data in HDFS [97], a distributed file system that resembles GFS [40]. HDFS triply replicates data in order to provide availability and tolerate failures. These properties free HBase to focus on higher-level database logic. Because HBase stores all its data in HDFS, the same machines are typically used to run both HBase and HDFS servers, thus improving locality. These clusters have three main types of machines: an *HBase master*, an *HDFS NameNode*, and many *worker* machines. Each worker runs two servers: an *HBase RegionServer* and an *HDFS DataNode*. HBase clients use a mapping generated by the HBase master to find the *one* RegionServer responsible for a given key. Similarly, an HDFS NameNode helps HDFS clients map a pathname and logical block number to the *three* DataNodes with replicas of that block.

3.2 Measurement Methodology

We now discuss trace collection and analysis (§3.2.1), simulation (§3.2.2), sensitivity of results (§3.2.3), and confidentiality (§3.2.4).

3.2.1 Trace Collection and Analysis

Prior Hadoop trace studies [21, 56] typically analyze default MapReduce or HDFS logs, which record coarse-grained file events (*e.g.*, creates and opens) but lack details about individual requests (*e.g.*, offsets and sizes). For our study, we build a new trace framework, HTFS (Hadoop Trace File System) to collect these details. Some data, though (*e.g.*, the contents of a write), is not recorded; this makes traces smaller and (more importantly) protects user privacy.

HBase accesses HDFS via the HDFS client library. We build HTFS, a new wrapper around the client library that records over 40 different HDFS calls. HDFS clients can be configured to use an arbitrary composition of a set of wrappers around the underlying library, so deploying HTFS is relatively straightforward. The ability to configure HDFS clients to use an arbitrary composition of wrappers has been used to extend HDFS with a variety of features (*e.g.*, some deployments wrap the client library with a client-side checksumming layer). FM is typically deployment with a client-side wrapper that enables fast failover upon NameNode failure [16]. We collect our traces by additionally wrapping the HDFS client used by the RegionServers on select machines with our HTFS layer. HTFS is publicly available with the Facebook branch of Hadoop.[1]

We collect our traces on a specially configured *shadow cluster* that receives the same requests as a production FM cluster. Facebook often uses shadow clusters to test new code before broad deployment. By tracing in

[1] `https://github.com/facebook/hadoop-20/blob/master/src/hdfs/org/apache/hadoop/hdfs/APITraceFileSystem.java`

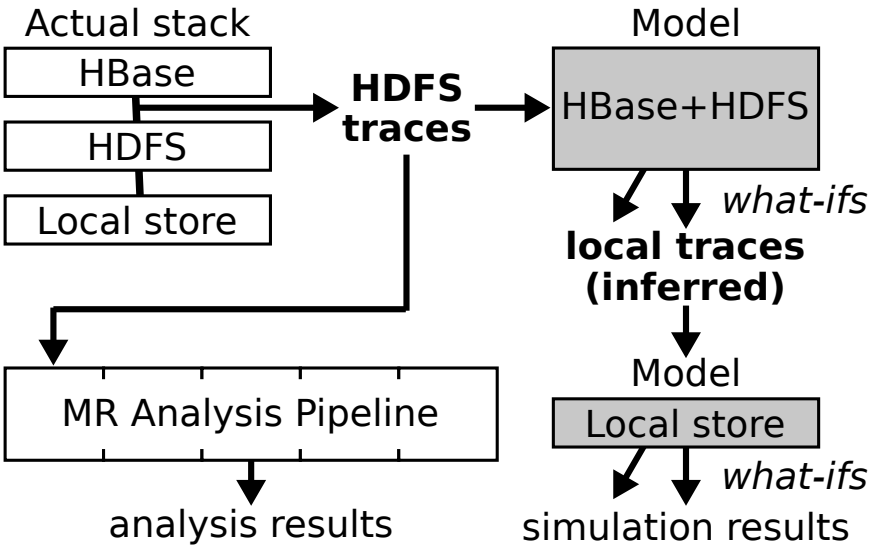

Figure 3.1: **Tracing, Analysis, and Simulation.**

an HBase/HDFS shadow cluster, we were able to study the real workload without imposing overheads on real users. For our study, we randomly selected nine worker machines, configuring each to use HTFS.

We collected traces for 8.3 days, starting June 7, 2013. We collected 116 GB of gzip-compressed traces, representing 5.2 billion recorded events and 71 TB of HDFS I/O. The machines each had 32 Xeon(R) CPU cores and 48 GB of RAM, 16.4 GB of which was allocated for the HBase cache (most memory is left to the file-system cache, as attempts to use larger caches in HBase cause JVM garbage-collection stalls). The HDFS workload is the product of a 60/34/6 get/put/delete ratio for HBase.

As Figure 3.1 shows, the traces enable both analysis and simulation. We analyzed our traces with a pipeline of 10 MapReduce jobs, each of which transforms the traces, builds an index, shards events, or outputs statistics. Complex dependencies between events require careful sharding for correctness. For instance, a stream-open event and a stream-write event must be in the same compute shard in order to correlate I/O with

file type. Furthermore, sharding must address the fact that different paths may refer to the same data (due to renames).

3.2.2 Modeling and Simulation

We evaluate changes to the storage stack via simulation. Our simulations are based on two models (illustrated in Figure 3.1): a model that determines how the HDFS I/O translates to local I/O and a model of local storage.

How HDFS I/O translates to local I/O depends on several factors, such as prior state, replication policy, and configurations. Making all these factors match the actual deployment would be difficult, and modeling what happens to be the current configuration is not particularly interesting. Thus, we opt for a model that is easy to understand and plausible (*i.e.*, it reflects a hypothetical policy and state that could reasonably be deployed).

Our HBase+HDFS model assumes the HDFS files in our traces are replicated by nine DataNodes that co-reside with the nine RegionServers we traced. The data for each RegionServer is replicated to one co-resident and two remote DataNodes. Our model ignores network latency: an HDFS write results in I/O on the remote DataNodes at the same instant in time. HDFS file blocks are 256 MB in size; thus, when a RegionServer writes a 1 GB HDFS file, our model translates that to the creation of twelve 256 MB local files (four per replica). Furthermore, 2 GB of network copies are counted for the remote replicas. This simplified model of replication could lead to errors for load balancing studies, but we believe little generality is lost for caching simulations and our other experiments. In production, all the replicas of a RegionServer's data may be remote (due to region re-assignment), causing additional network I/O; however, long-running FM-HBase clusters tend to converge over time to the pattern we simulate.

The HDFS+HBase model's output is the input for our local-store model. Each local store is assumed to have an HDFS DataNode, a set of disks (each with its own file system and disk scheduler), a RAM cache, and possibly an SSD. When the simulator processes a request, a balancer module representing the DataNode logic directs the request to the appropriate disk. The file system for that disk checks the RAM and flash caches; upon a miss, the request is passed to a disk scheduler for re-ordering.

The scheduler in our local-store module switches between files using a round-robin policy (1 MB slice). The C-SCAN policy [8] is then used to choose between multiple requests to the same file. The scheduler dispatches requests to a disk module that determines latency. Requests to different files are assumed to be distant, and so require a 10ms seek. Requests to adjacent offsets of the same file, however, are assumed to be adjacent on disk, so blocks are transferred at 100 MB/s. Finally, we assume some locality between requests to non-adjacent offsets in the same file; for these, the seek time is $\min\{10\text{ms}, \text{distance}/(100\text{MB/s})\}$.

3.2.3 Sensitivity Analysis

We now address three validity questions: *does ignoring network latency skew our results? Did we run our simulations long enough? Are simulation results from a single representative machine meaningful?*

First, we explore our assumption about constant network latency by adding random jitter to the timing of requests and observing how important statistics change. Table 3.1 shows how much error results by changing request issue times by a uniform-random amount. Errors are very small for 1ms jitter (at most 1.3% error). Even with a 10ms jitter, the worst error is 6.6%. Second, in order to verify that we ran the simulations long enough, we measure how the statistics would have been different if we had finished our simulations 2 or 4 days earlier (instead of using the full 8.3 days of traces). The differences are worse than for jitter, but are still

		jitter ms			*finish day*		*sample*
statistic	**baseline**	**1**	**5**	**10**	**-2**	**-4**	**median**
FS reads MB/min	576	0.0	0.0	0.0	-3.4	-0.6	-4.2
FS writes MB/min	447	0.0	0.0	0.0	-7.7	-11.5	-0.1
RAM reads MB/min	287	-0.0	0.0	0.0	-2.6	-2.4	-6.2
RAM writes MB/min	345	0.0	-0.0	-0.0	-3.9	1.1	-2.4
Disk reads MB/min	345	-0.0	0.0	0.0	-3.9	1.1	-2.4
Disk writes MB/min	616	-0.0	1.3	1.9	-5.3	-8.3	-0.1
Net reads MB/min	305	0.0	0.0	0.0	-8.7	-18.4	-2.8
Disk reqs/min	275.1K	0.0	0.0	0.0	-4.6	-4.7	-0.1
(user-read)	65.8K	0.0	-0.0	-0.0	-2.9	-0.8	-4.3
(log)	104.1K	0.0	0.0	0.0	1.6	1.3	-1.0
(flush)	4.5K	0.0	0.0	0.0	1.2	0.4	-1.3
(compact)	100.6K	-0.0	-0.0	-0.0	-12.2	-13.6	-0.1
Disk queue ms	6.17	-0.4	-0.5	-0.0	-3.2	0.6	-1.8
(user-read)	12.3	0.1	-0.8	-1.8	-0.2	2.7	1.7
(log)	2.47	-1.3	-1.1	0.6	-4.9	-6.4	-6.0
(flush)	5.33	0.3	0.0	-0.3	-2.8	-2.6	-1.0
(compact)	6.0	-0.6	0.0	2.0	-3.5	2.5	-6.4
Disk exec ms	0.39	0.1	1.0	2.5	1.0	2.0	-1.4
(user-read)	0.84	-0.1	-0.5	-0.7	-0.0	-0.1	-1.2
(log)	0.26	0.4	3.3	6.6	-2.1	-1.7	0.0
(flush)	0.15	-0.3	0.7	3.2	-1.1	-0.9	-0.8
(compact)	0.24	-0.0	2.1	5.2	4.0	4.8	-0.3

Table 3.1: **Statistic Sensitivity.** *The first column group shows important statistics and their values for a representative machine. Other columns show how these values would change (as percentages) if measurements were done differently. Low percentages indicate a statistic is robust.*

usually small, and are at worst 18.4% for network I/O.

Finally, we evaluate whether it is reasonable to pick a single representative instead of running our experiments for all nine machines in our sample. Running all our experiments for a single machine alone takes about 3 days on a 24-core machine with 72 GB of RAM, so basing our results on a representative is desirable. The final column of Table 3.1 compares the difference between statistics for our representative machine and the median of statistics for all nine machines. Differences are quite small and are never greater than 6.4%, so we use the representative for the remainder of our simulations (trace-analysis results, however, will be based on all nine machines).

3.2.4 Confidentiality

In order to protect user privacy, our traces only contain the sizes of data (*e.g.*, request and file sizes), but never actual data contents. Our tracing code was carefully reviewed by Facebook employees to ensure compliance with Facebook privacy commitments. We also avoid presenting commercially-sensitive statistics, such as would allow estimation of the number of users of the service. While we do an in-depth analysis of the I/O patterns on a sample of machines, we do not disclose how large the sample is as a fraction of all the FM clusters. Much of the architecture we describe is open source.

3.3 Workload Behavior

We now characterize the FM workload with four questions: *what are the major causes of I/O at each layer of the stack (§3.3.1)? How much I/O and space is required by different types of data (§3.3.2)? How large are files, and does file size predict file lifetime (§3.3.3)?* And *do requests exhibit patterns such as locality or sequentiality (§3.3.4)?*

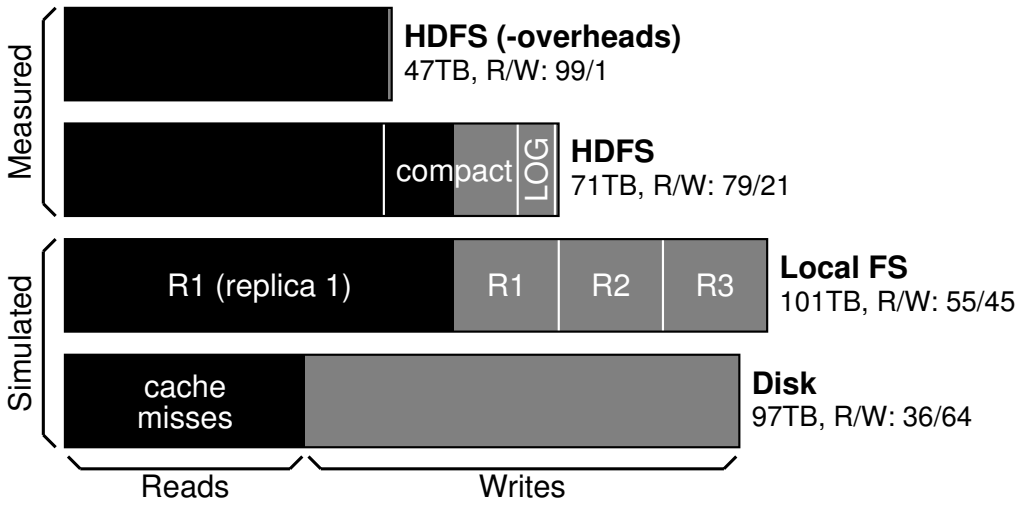

Figure 3.2: **I/O Across Layers.** *The horizontal length of the bars represents bytes of I/O at various levels of the storage stack. Black sections represent reads and gray sections represent writes. The top two bars indicate HDFS I/O as measured directly in the traces. The bottom two bars indicate local I/O at the file-system and disk layers as inferred via simulation.*

3.3.1 Multilayer Overview

We begin by considering the number of reads and writes at each layer of the stack in Figure 3.2. At a high level, FM issues `put()` and `get()` requests to HBase. The `put` data accumulates in buffers, which are occasionally flushed to *HFiles* (HDFS files containing sorted key-value pairs and indexing metadata). Thus, `get` requests consult the write buffers as well as the appropriate HFiles in order to retrieve the most up-to-date value for a given key. This core I/O (`put`-flushes and `get`-reads) is shown in the first bar of Figure 3.2; the 47 TB of I/O is 99% reads.

In addition to the core I/O, HBase also does logging (for durability) and compaction (to maintain a read-efficient layout) as shown in the second bar. Writes account for most of these overheads, so the R/W (read-/write) ratio decreases to 79/21. Flush data is compressed but log data is not, so logging causes 10x more writes even though the same data is

both logged and flushed. Preliminary experiments with log compression [99] have reduced this ratio to 4x. However, flushes have a fundamental advantage over logs with regard to compression. Flushed data can be compressed in written in large chunks, so there is more potential redundancy to be exploited by a compression algorithm. In contrast, log writes must complete in smaller increments as `puts` arrive, so it is inherently more difficult to compress.

Compaction causes about 17x more writes than flushing does. Flushes of new data forces compaction with existing data in order to prevent reads from becoming increasingly expensive, so we conclude there as a 17x compaction amplification for writes. FM stores very large objects (*e.g.*, image attachments) in Haystack [12] for this reason. FM is a very read-heavy HBase workload within Facebook, so it is tuned to compact aggressively. Compaction makes reads faster by merge-sorting many small HFiles into fewer big HFiles, thus reducing the number of files a `get` must check.

FM tolerates failures by replicating data with HDFS. Thus, writing an HDFS block involves writing three local files and two network transfers. The third bar of Figure 3.2 shows how this tripling further reduces the R/W ratio to 55/45. Furthermore, OS caching prevents some of these file-system reads from hitting disk. With a 30 GB cache, the 56 TB of reads at the file-system level cause only 35 TB of reads at the disk level, as shown in the fourth bar. Also, very small file-system writes cause 4 KB-block disk writes, so writes are increased at the disk level. Because of these factors, writes represent 64% of disk I/O.

Figure 3.3 gives a similar layered overview, but for the amount of data stored at each layer of the stack rather than for I/O. The first bar shows 3.9 TB of HDFS data received some core I/O during tracing (data deleted during tracing is not counted). Nearly all this data was read and a small portion written. The second bar also includes data which was accessed only by non-core I/O; non-core data is several times bigger than core data.

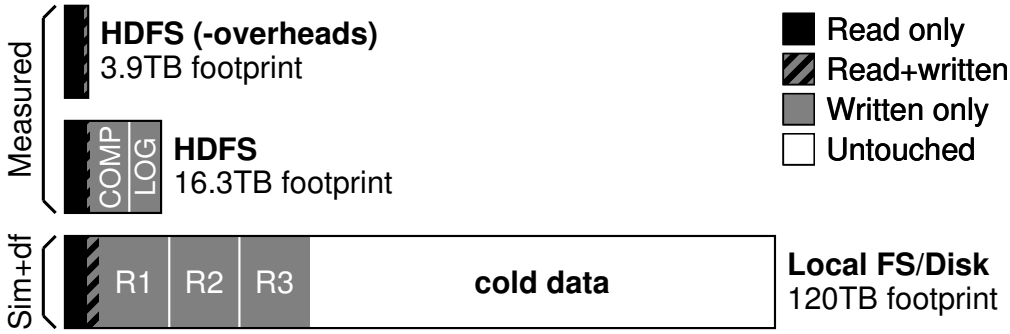

Figure 3.3: **Data Across Layers.** *The horizontal length of the bars represents bytes of logical data at various levels of the storage stack. This is the same as Figure 3.2 but for data size instead of I/O. COMP is compaction.*

The third bar shows how much data is touched at the local level during tracing. This bar also shows *untouched* data; we estimate[2] this by subtracting the amount of data we infer was touched due to HDFS I/O from the disk utilization (measured with `df`). Most of the 120 TB of data is very cold; only a third is accessed over the 8-day period.

Conclusion: FM is very read-heavy, but logging, compaction, replication, and caching amplify write I/O, causing writes to dominate disk I/O. We also observe that while the HDFS dataset accessed by core I/O is relatively small, on disk the dataset is very large (120 TB) and very cold (two thirds is never touched). Thus, architectures to support this workload should consider its hot/cold nature.

3.3.2 Data Types

We now study the types of data FM stores. Each user's data is stored in a single HBase row; this prevents the data from being split across different RegionServers. New data for a user is added in new columns within the

[2]the RegionServers in our sample store some data on DataNodes outside our sample (and vice versa), so this is a sample-based estimate rather than a direct correlation of HDFS data to disk data

Family	Description
Actions	Log of user actions and message contents
MessageMeta	Metadata per message (*e.g.*, isRead and subject)
ThreadMeta	Metadata per thread (*e.g.*list of participants)
PrefetchMeta	Privacy settings, contacts, mailbox summary, etc.
Keywords	Word-to-message map for search and typeahead
ThreaderThread	Thread-to-message mapping
ThreadingIdIdx	Map between different types of message IDs
ActionLogIdIdx	Also a message-ID map (like ThreadingIdIdx)

Table 3.2: **Schema.** *HBase Column Families are Described.*

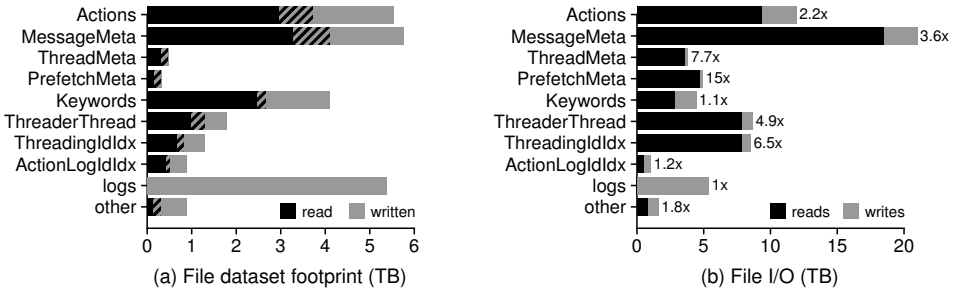

Figure 3.4: **File Types.** *Left: all accessed HDFS file data is broken down by type. Bars further show whether data was read, written, or both. Right: I/O is broken down by file type and read/write. Bar labels indicate the I/O-to-data ratio.*

row. Related columns are grouped into families, which are defined by the FM schema (summarized in Table 3.2).

The *Actions* family is a log built on top of HBase, with different log records stored in different columns; *addMsg* records contain actual message data while other records (*e.g.*, *markAsRead*) record changes to metadata state. Getting the latest state requires reading a number of recent records in the log. To cap this number, a metadata snapshot (a few hundred bytes) is sometimes written to the *MessageMeta* family. Because Facebook chat is built over messages, metadata objects are large relative to many messages (*e.g.*, "hey, whasup?"). Thus, writing a change to *Actions* is generally much cheaper than writing a full metadata object to *Mes-*

sageMeta. Other metadata is stored in *ThreadMeta* and *PrefetchMeta* while *Keywords* is a keyword-search index and *ThreaderThread*, *ThreadingIdIdx*, and *ActionLogIdIdx* are other indexes.

Figure 3.4a shows how much data of each type is accessed at least once during tracing (including later-deleted data); a total (sum of bars) of 26.5 TB is accessed. While actual messages (*i.e.*, *Actions*) take significant space, helper data (*e.g.*, metadata, indexes, and logs) takes much more. We also see that little data is both read and written, suggesting that writes should be cached selectively (if at all). Figure 3.4b reports the I/O done for each type. We observe that some families receive much more I/O per data, *e.g.*, an average data byte of *PrefetchMeta* receives 15 bytes of I/O whereas a byte of *Keywords* receives only 1.1.

Conclusion: FM uses significant space to store messages and does a significant amount of I/O on these messages; however, both space and I/O are dominated by helper data (*i.e.*, metadata, indexes, and logs). Relatively little data is both written and read during tracing; this suggests caching writes is of little value.

3.3.3 File Size

GFS (the inspiration for HDFS) assumed that "multi-GB files are the common case, and should be handled efficiently" [40]. Other workload studies confirm this, *e.g.*, MapReduce inputs were found to be about 23 GB at the 90th percentile (Facebook in 2010) [21]. We now revisit the assumption that HDFS files are large.

Figure 3.5 shows, for each file type, a distribution of file sizes (about 862 thousand files appear in our traces). Most files are small; for each family, 90% are smaller than 15 MB. However, a handful are so large as to skew averages upwards significantly, *e.g.*, the average MessageMeta file is 293 MB.

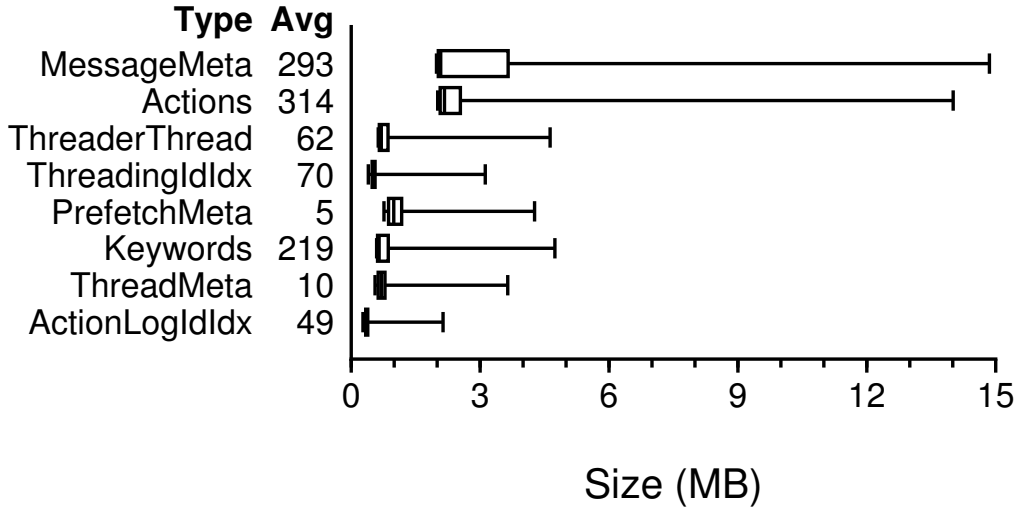

Figure 3.5: **File-Size Distribution.** *This shows a box-and-whiskers plot of file sizes. The whiskers indicate the 10th and 90th percentiles. On the left, the type of file and average size is indicated. Log files are not shown, but have an average size of 218 MB with extremely little variance.*

Although most files are very small, compaction should quickly replace these small files with a few large, long-lived files. We divide files created during tracing into small (0 to 16 MB), medium (16 to 64 MB), and large (64 MB+) categories. 94% of files are small, 2% are medium, and 4% are large; however, large files contain 89% of the data. Figure 3.6 shows the distribution of file lifetimes for each category. 17% of small files are deleted within less than a minute, and very few last more than a few hours; about half of medium files, however, last more than 8 hours. Only 14% of the large files created during tracing were also deleted during tracing.

Conclusion: Traditional HDFS workloads operate on very large files. While most FM data lives in large, long-lived files, most files are small and short-lived. This has metadata-management implications; HDFS manages all file metadata with a single NameNode because the data-to-metadata ratio is assumed to be high. For FM, this assumption does not hold; per-

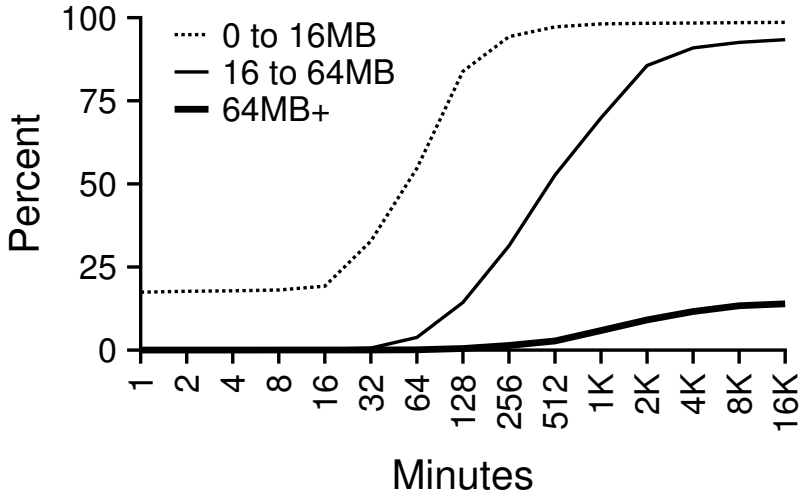

Figure 3.6: **Size/Life Correlation.** *Each line is a CDF of lifetime for created files of a particular size. Not all lines reach 100% as some files are not deleted during tracing.*

haps distributing HDFS metadata management should be reconsidered.

3.3.4 I/O Patterns

We explore three relationships between different read requests: temporal locality, spatial locality, and sequentiality. We use a new type of plot, a *locality map*, that describes all three relationships at once. Figure 3.7 shows a locality map for FM reads. The data shows how often a read was *recently* preceded by a *nearby* read, for various thresholds on "recent" and "nearby". Each line is a hit-ratio curve, with the x-axis indicating how long items are cached. Different lines represent different levels of prefetching, *e.g.*, the 0-line represents no prefetching, whereas the 1 MB-line means data 1 MB before and 1 MB after a read is prefetched.

Line shape describes *temporal locality*, *e.g.*, the 0-line gives a distribution of time intervals between different reads to the same data. Reads are

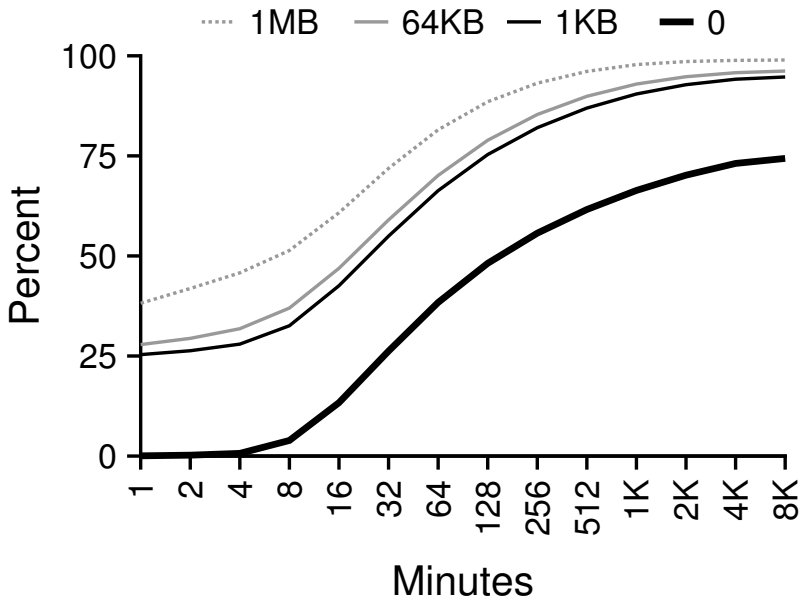

Figure 3.7: **Reads: Locality Map.** *This plot shows how often a read was recently preceded by a nearby read, with time-distance represented along the x-axis and offset-distance represented by the four lines.*

almost never preceded by a prior read to the same data in the past four minutes; however, 26% of reads are preceded within the last 32 minutes. Thus, there is significant temporal locality (*i.e.*, reads are near each other with respect to time), and additional caching should be beneficial. The locality map also shows there is little *sequentiality*. A highly sequential pattern would show that many reads were recently preceded by I/O to nearby offsets; here, however, the 1 KB-line shows only 25% of reads were preceded by I/O to very nearby offsets within the last minute. Thus, over 75% of reads are random. The distances between the lines of the locality map describe *spatial locality*. The 1 KB-line and 64 KB-line are very near each other, indicating that (except for sequential I/O) reads are rarely preceded by other reads to nearby offsets. This indicates very low spatial locality (*i.e.*, reads are far from each other with respect to offset), and additional prefetching is unlikely to be helpful.

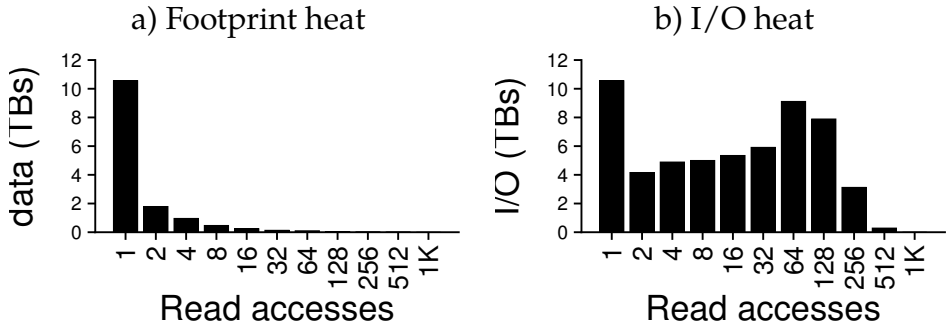

Figure 3.8: **Read Heat.** *In both plots, bars show a distribution across different levels of read heat (i.e., the number of times a byte is read). The left shows a distribution of the dataset (so the bars sum to the dataset size, included deleted data), and the right shows a distribution of I/O to different parts of the dataset (so the bars sum to the total read I/O).*

To summarize the locality map, the main pattern reads exhibit is temporal locality (there is little sequentiality or spatial locality). High temporal locality implies a significant portion of reads are "repeats" to the same data. We explore this repeated-access pattern further in Figure 3.8a. The bytes of HDFS file data that are read during tracing are distributed along the x-axis by the number of reads. The figure shows that most data (73.7%) is read only once, but 1.1% of the data is read at least 64 times. Thus, repeated reads are not spread evenly, but are concentrated on a small subset of the data.

Figure 3.8b shows how many bytes are read for each of the categories of Figure 3.8a. While 19% of the reads are to bytes which are only read once, most I/O is to data which is accessed many times. Such bias at this level is surprising considering that all HDFS I/O has missed two higher-level caches (an application cache and the HBase cache). Caches are known to lessen I/O to particularly hot data, *e.g.*, a multilayer photo-caching study found caches cause "distributions [to] flatten in a significant way" [53]. The fact that bias remains despite caching suggests the working set may be too large to fit in a small cache; a later section (§3.4.1)

H/W	Cost	Failure rate	Performance
HDD	$100/disk	4% AFR [37]	10ms/seek, 100 MB/s
RAM	$5.0/GB	4% AFR (8 GB)	0 latency
Flash	$0.8/GB	10K P/E cycles	0.5ms latency

Table 3.3: **Cost Model.** *Our assumptions about hardware costs, failure rates, and performance are presented. For disk and RAM, we state an AFR (annual failure rate), assuming uniform-random failure each year. For flash, we base replacement on wear and state program/erase cycles.*

shows this to be the case.

Conclusion: At the HDFS level, FM exhibits relatively little sequentiality, suggesting high-bandwidth, high-latency storage mediums (*e.g.*, disk) are not ideal for serving reads. The workload also shows very little spatial locality, suggesting additional prefetching would not help, possibly because FM already chooses for itself what data to prefetch. However, despite application-level and HBase-level caching, some of the HDFS data is particularly hot; thus, additional caching could help.

3.4 Tiered Storage: Adding Flash

We now make a case for adding a flash tier to local machines. FM has a very large, mostly cold dataset (§3.3.1); keeping all this data in flash would be wasteful, costing upwards of $10K/machine[3]. We evaluate the two alternatives: use some flash or no flash. We explore tradeoffs between various configurations via simulation (§3.2.2), using the assumptions summarized in Table 3.3 unless otherwise specified.

We consider four questions: *how much can we improve performance without flash, by spending more on RAM or disks (§3.4.1)? What policies utilize a tiered RAM/flash cache best (§3.4.2)? Is flash better used as a cache to absorb*

[3]at $0.80/GB, storing 13.3 TB (120 TB split over 9 machines) in flash would cost $10,895/machine.

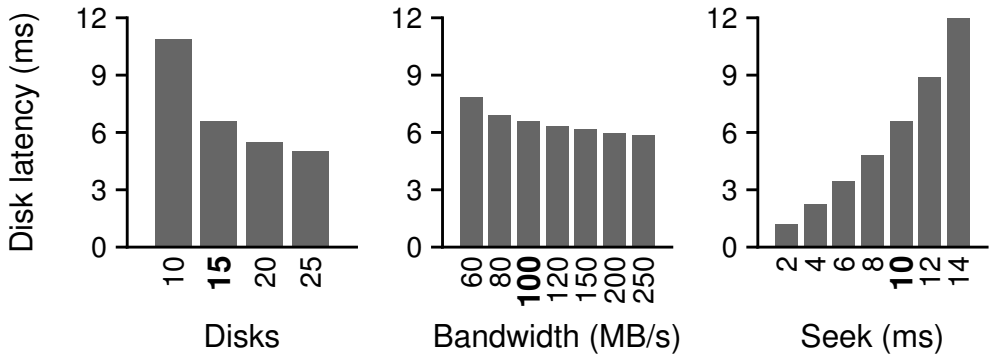

Figure 3.9: **Disk Performance.** *The figure shows the relationship between disk characteristics and the average latency of disk requests. As a default, we use 15 disks with 100 MB/s bandwidth and 10ms seek time. Each of the plots varies one of the characteristics, keeping the other two fixed.*

reads or as a buffer to absorb writes (§3.4.3)? And ultimately, *is the cost of a flash tier justifiable (§3.4.4)?*

3.4.1 Performance without Flash

Can buying faster disks or more disks significantly improve FM performance? Figure 3.9 presents average disk latency as a function of various disk factors. The first plot shows that for more than 15 disks, adding more disks has quickly diminishing returns. The second shows that higher-bandwidth disks also have relatively little advantage, as anticipated by the highly-random workload observed earlier (§3.3.4). However, the third plot shows that latency is a major performance factor.

The fact that lower latency helps more than having additional disks suggests the workload has relatively little parallelism, *i.e.*, being able to do a few things quickly is better than being able to do many things at once. Unfortunately, the 2-6ms disks we simulate are unrealistically fast, having no commercial equivalent. Thus, although significant disk capacity is needed to store the large, mostly cold data, reads are better served

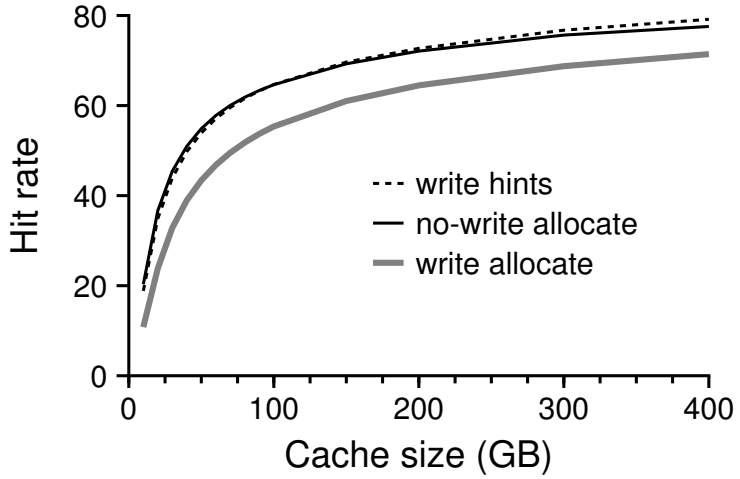

Figure 3.10: **Cache Hit Rate.** *The relationship between cache size and hit rate is shown for three policies.*

by a low-latency medium (*e.g.*, RAM or flash).

Thus, we ask, *can the hot data fit comfortably in a pure-RAM cache?* We measure hit rate for cache sizes in the 10-400 GB range. We also try three different LRU policies: *write allocate*, *no-write allocate*, and *write hints*. All three are write-through caches, but differ regarding whether written data is cached. Write allocate adds all write data, no-write allocate adds no write data, and the hint-based policy takes suggestions from HBase and HDFS. In particular, a written file is only cached if (a) the local file is a primary replica of the HDFS block, and (b) the file is either flush output (as opposed to compaction output) or is likely to be compacted soon.

Figure 3.10 shows, for each policy, that the hit rate increases significantly as the cache size increases up until about 200 GB, where it starts to level off (but not flatten); this indicates the working set is very large. Earlier (§3.3.2), we found little overlap between writes and reads and concluded that written data should be cached selectively if at all. Figure 3.10 confirms: caching all writes is the worst policy. Up until about 100 GB, "no-write allocate" and "write hints" perform about equally well. Beyond 100 GB, hints help, but only slightly. We use no-write allocate through-

out the remainder of the paper because it is simple and provides decent performance.

Conclusion: The FM workload exhibits relatively little sequentiality or parallelism, so adding more disks or higher-bandwidth disks is of limited utility. Fortunately, the same data is often repeatedly read (§3.3.4), so a very large cache (*i.e.*, a few hundred GBs in size) can service nearly 80% of the reads. The usefulness of a very large cache suggests that storing at least some of the hot data in flash may be most cost effective. We evaluate the cost/performance tradeoff between pure-RAM and hybrid caches in a later section (§3.4.4).

3.4.2 Flash as Cache

In this section, we use flash as a second caching tier beneath RAM. Both tiers independently are LRU. Initial inserts are to RAM, and RAM evictions are inserted into flash. We evaluate exclusive cache policies. Thus, upon a flash hit, we have two options: the *promote policy* (PP) repromotes the item to the RAM cache, but the *keep policy* (KP) keeps the item at the flash level. PP gives the combined cache LRU behavior. The idea behind KP is to limit SSD wear by avoiding repeated promotions and evictions of items between RAM and flash.

Figure 3.11 shows the hit rates for twelve flash/RAM mixes. For example, the middle plot shows what the hit rate is when there is 30 GB of RAM: without any flash, 45% of reads hit the cache, but with 60 GB of flash, about 63% of reads hit in either RAM or flash (regardless of policy). The plots show that across all amounts of RAM and flash, the number of reads that hit in "any" cache differs very little between policies. However, PP causes significantly more of these hits to go to RAM; thus, PP will be faster because RAM hits are faster than flash hits.

We now test our hypothesis that, in trade for decreasing RAM hits, KP improves flash lifetime. We compute lifetime by measuring flash writes,

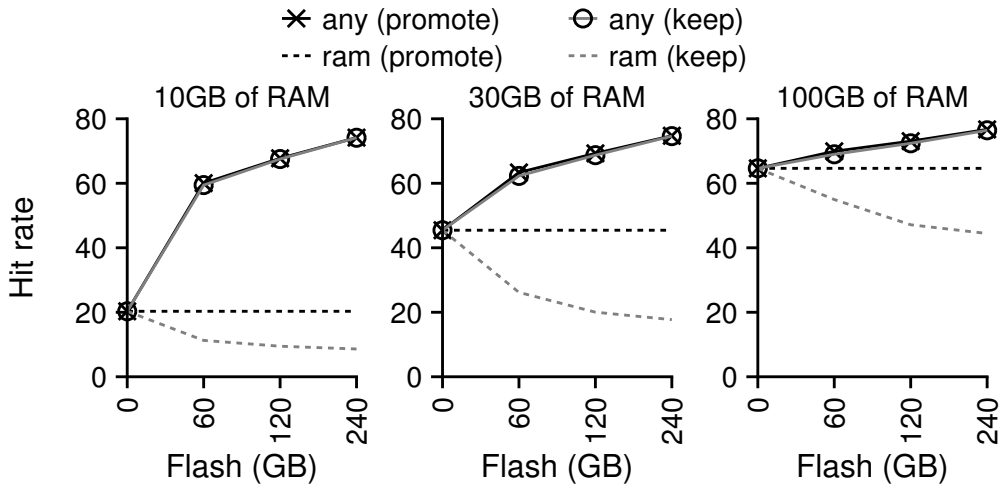

Figure 3.11: **Tiered Hit Rates.** *Overall hit rate (any) is shown by the solid lines for the promote and keep policies. The results are shown for varying amounts of RAM (different plots) and varying amounts of flash (x-axis). RAM hit rates are indicated by the dashed lines.*

assuming the FTL provides even wear leveling, and assuming the SSD supports 10K program/erase cycles. Figure 3.12 reports flash lifetime as the amount of flash varies along the x-axis.

The figure shows that having more RAM slightly improves flash lifetime. This is because flash writes occur upon RAM evictions, and evictions will be less frequent with ample RAM. Also, as expected, KP often doubles or triples flash lifetime, *e.g.*, with 10 GB of RAM and 60 GB of flash, using KP instead of PP increases lifetime from 2.5 to 5.2 years. The figure also shows that flash lifetime increases with the amount of flash. For PP, the relationship is perfectly linear. The number of flash writes equals the number of RAM evictions, which is independent of flash size; thus, if there is twice as much flash, each block of flash will receive exactly half as much wear. For KP, however, the flash lifetime increases superlinearly with size; with 10 GB of RAM and 20 GB of flash, the years-to-GB ratio is 0.06, but with 240 GB of flash, the ratio is 0.15. The relationship is superlinear because additional flash absorbs more reads, causing fewer

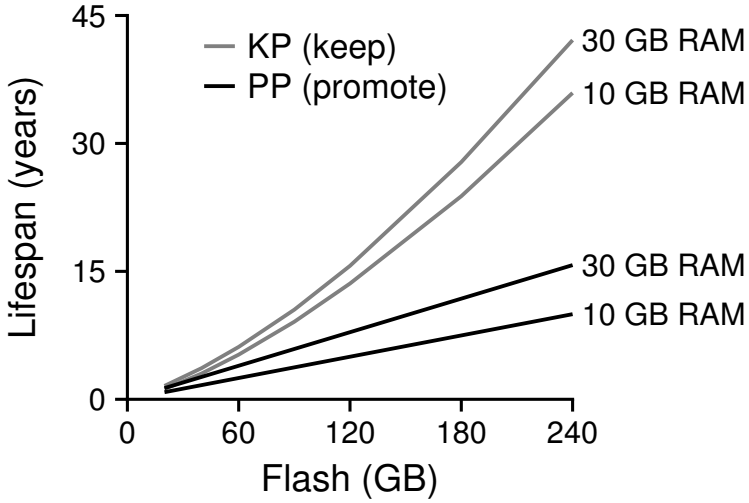

Figure 3.12: **Flash Lifetime.** *The relationship between flash size and flash lifetime is shown for both the keep policy (gray lines) and promote policy (black lines). There are two lines for each policy (10 or 30 GB RAM).*

RAM inserts, causing fewer RAM evictions, and ultimately causing fewer flash writes. Thus, doubling the flash size decreases total flash writes in addition to spreading the writes over twice as many blocks.

Flash caches have an additional advantage: crashes do not cause cache contents to be lost. We quantify this benefit by simulating four crashes at different times and measuring changes to hit rate. Figure 3.13 shows the results of two of these crashes for 100 GB caches with different flash-to-RAM ratios (using PP). Even though the hottest data will be in RAM, keeping some data in flash significantly improves the hit rate after a crash. The examples also show that it can take 4-6 hours to fully recover from a crash. We quantify the total recovery cost in terms of additional disk reads (not shown). Whereas crashing with a pure-RAM cache on average causes 26 GB of additional disk I/O, crashing costs only 10 GB for a hybrid cache which is 75% flash.

Conclusion: Adding flash to RAM can greatly improve the caching hit rate; furthermore (due to persistence) a hybrid flash/RAM cache can

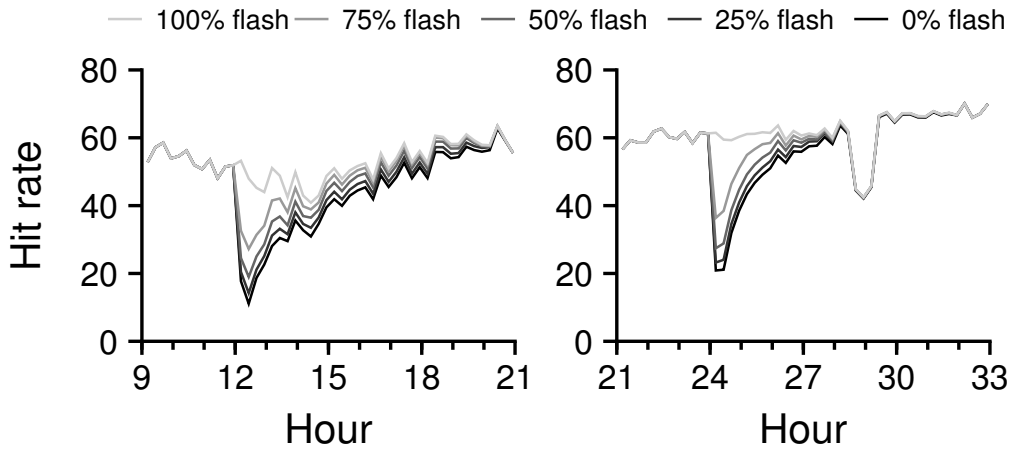

Figure 3.13: **Crash Simulations.** *The plots show two examples of how crashing at different times affects different 100 GB tiered caches, some of which are pure flash, pure RAM, or a mix. Hit rates are unaffected when crashing with 100% flash.*

eliminate half of the extra disk reads that usually occur after a crash. However, using flash raises concerns about wear. Shuffling data between flash and RAM to keep the hottest data in RAM improves performance but can easily decrease SSD lifetime by a factor of 2x relative to a wear-aware policy. Fortunately, larger SSDs tend to have long lifetimes for FM, so wear may be a small concern (*e.g.*, 120 GB+ SSDs last over 5 years regardless of policy).

3.4.3 Flash as Buffer

Another advantage of flash is that (due to persistence) it has the potential to reduce disk writes as well as reads. We saw earlier (§3.3.3) that files tend to be either small and short-lived or big and long-lived, so one strategy would be to store small files in flash and big files on disk.

HDFS writes are considered durable once the data is in memory on every DataNode (but not necessarily on disk), so buffering in flash would

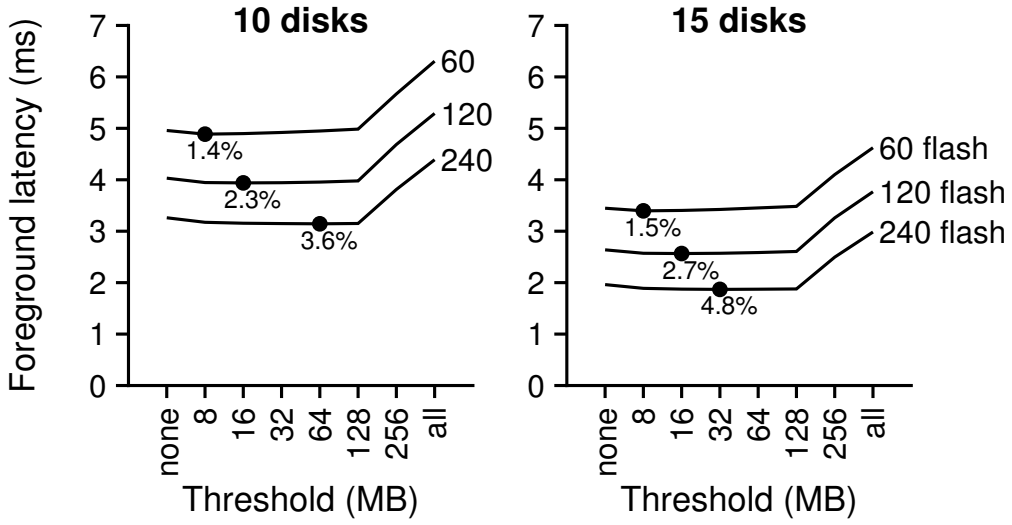

Figure 3.14: **Flash Buffer.** *We measure how different file-buffering policies impact foreground requests with two plots (for 10 or 15 disks) and three lines (60, 120, or 240 GB of flash). Different points on the x-axis represent different policies. The optimum point on each line is marked, showing improvement relative to the latency when no buffering is done.*

not actually improve HDFS write performance. However, decreasing disk writes by buffering the output of *background activities* (*e.g.*, flushes and compaction) indirectly improves *foreground* performance. Foreground activity includes any local requests which could block an HBase request (*e.g.*, a `get`). Reducing background I/O means foreground reads will face less competition for disk time. Thus, we measure how buffering files written by background activities affects foreground latencies.

Of course, using flash as a write buffer has a cost, namely less space for caching hot data. We evaluate this tradeoff by measuring performance when using flash to buffer only files which are beneath a certain size. Figure 3.14 shows how latency corresponds to the policy. At the left of the x-axis, writes are never buffered in flash, and at the right of the x-axis, all writes are buffered. Other x-values represent thresholds; only files smaller than the threshold are buffered. The plots show that buffering

all or most of the files results in very poor performance. Below 128 MB, though, the choice of how much to buffer makes little difference. The best gain is just a 4.8% reduction in average latency relative to performance when no writes are buffered.

Conclusion: Using flash to buffer all writes results in much worse performance than using flash only as a cache. If flash is used for both caching and buffering, and if policies are tuned to only buffer files of the right size, then performance can be slightly improved. We conclude that these small gains are probably not worth the added complexity, so flash should be for caching only.

3.4.4 Is Flash Worth the Money?

Adding flash to a system can, if used properly, only improve performance, so the interesting question is, given that we want to buy performance with money, *should we buy flash, or something else?* We approach this question by making assumptions about how fast and expensive different storage mediums are, as summarized in Table 3.3. We also state assumptions about component failure rates, allowing us to estimate operating expenditure.

We evaluate 36 systems, with three levels of RAM (10 GB, 30 GB, or 100 GB), four levels of flash (none, 60 GB, 120 GB, or 240 GB), and three levels of disk (10, 15, or 20 disks). Flash and RAM are used as a hybrid cache with the promote policy (§3.4.2). For each system, we compute the capex (capital expenditure) to initially purchase the hardware and determine via simulation the foreground latencies (defined in §3.4.3). Figure 3.15 shows the cost/performance of each system. 11 of the systems (31%) are highlighted; these are the only systems that one could justify buying (i.e., they are Pareto optimal). Each of the other 25 systems is both slower and more expensive than one of these 11 justifiable systems. Over half of the justifiable systems have maximum flash. It is worth noting

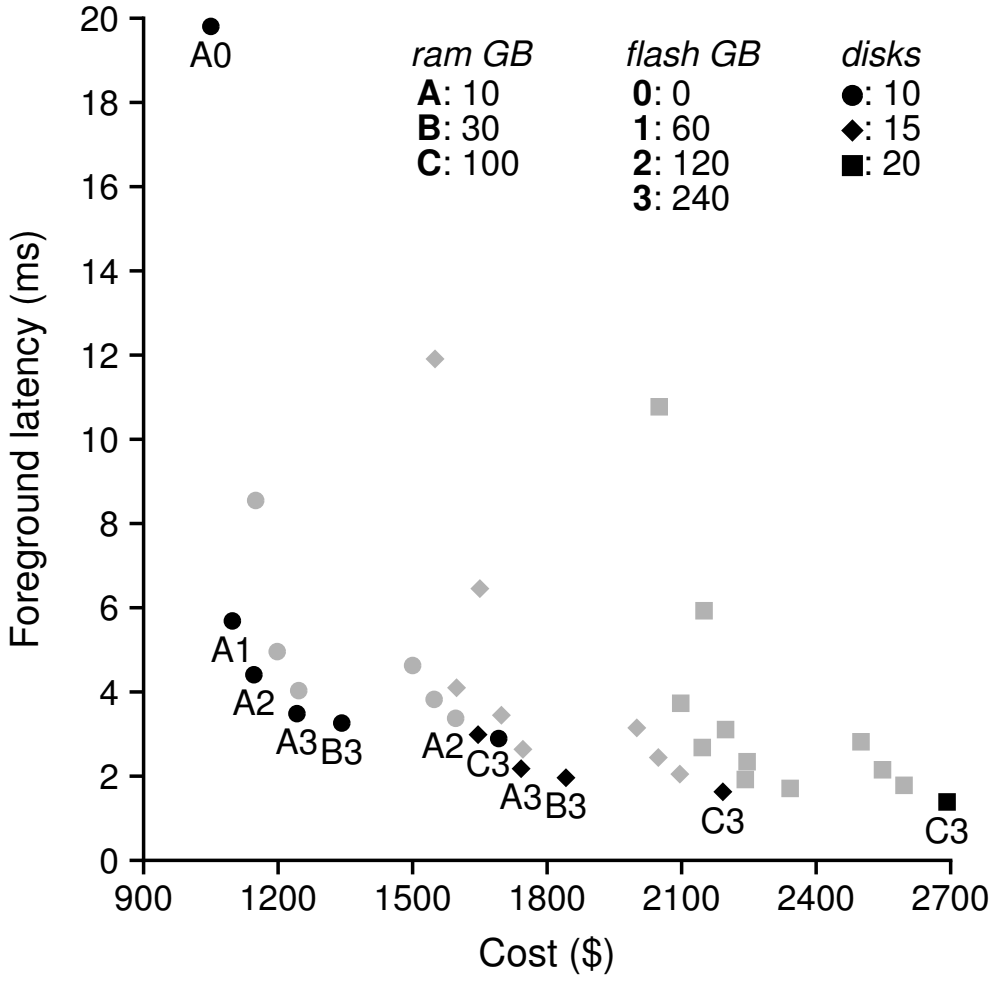

Figure 3.15: **Capex/Latency Tradeoff.** *We present the cost and performance of 36 systems, representing every combination of three RAM levels, four flash levels, and three disk levels. Combinations which present unique tradeoffs are black and labeled; unjustifiable systems are gray and unlabeled.*

that the systems with less flash are justified by low cost, not good performance. With one exception (15-disk A2), all systems with less than the maximum flash have the minimum number of disks and RAM. We observe that flash can greatly improve performance at very little cost. For example, A1 has a 60 GB SSD but is otherwise the same as A0. With 10

disks, A1 costs only 4.5% more but is 3.5x faster. We conclude that if performance is to be bought, then (within the space we explore) flash should be purchased first.

We also consider expected opex (operating expenditure) for replacing hardware as it fails, and find that replacing hardware is relatively inexpensive compared to capex (not shown). Of the 36 systems, opex is at most $90/year/machine (for the 20-disk C3 system). Furthermore, opex is never more than 5% of capex. For each of the justifiable flash-based systems shown in Figure 3.15, we also do simulations using KP for flash hits. KP decreased opex by 4-23% for all flash machines while increasing latencies by 2-11%. However, because opex is low in general, the savings are at most $14/year/machine.

Conclusion: Not only does adding a flash tier to the FM stack greatly improve performance, but it is the most cost-effective way of improving performance. In some cases, adding a small SSD can triple performance while only increasing monetary costs by 5%.

3.5 Layering: Pitfalls and Solutions

The FM stack, like most storage, is a composition of other systems and subsystems. Some composition is horizontal; for example, FM stores small data in HBase and large data in Haystack (§3.3.1). In this section, we focus instead on the vertical composition of layers, a pattern commonly used to manage and reduce software complexity. We discuss different ways to organize storage layers (§3.5.1), how to reduce network I/O by bypassing the replication layer (§3.5.2), and how to reduce the randomness of disk I/O by adding special HDFS support for HBase logging (§3.5.3).

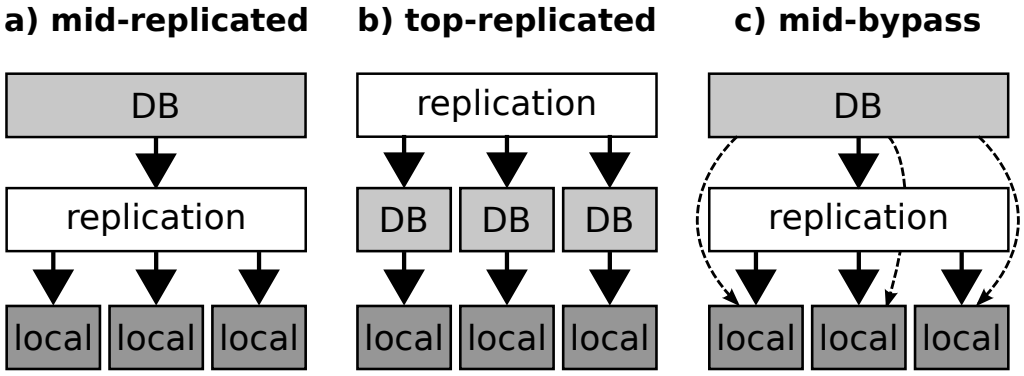

Figure 3.16: **Layered Architectures.** *The HBase architecture (mid-replicated) is shown, as well as two alternatives. Top-replication reduces network I/O by co-locating database computation with database data. The mid-bypass architecture is similar to mid-replication, but provides a mechanism for bypassing the replication layer for efficiency.*

3.5.1 Layering Background

Three important layers are the *local layer* (*e.g.*, disks, local file systems, and a DataNode), the *replication layer* (*e.g.*, HDFS), and the *database layer* (*e.g.*, HBase). FM composes these in a *mid-replicated* pattern (Figure 3.16a), with the database at the top of the stack and the local stores at the bottom. The merit of this architecture is simplicity. The database can be built with the assumption that underlying storage, because it is replicated, will be available and never lose data. The replication layer is also relatively simple, as it deals with data in its simplest form (*i.e.*, large blocks of opaque data). Unfortunately, mid-replicated architectures separate computation from data. Computation (*e.g.*, database operations such as compaction) can only be co-resident with at most one replica, so all writes involve network transfers.

Top-replication (Figure 3.16b) is an alternative approach used by the Salus storage system [112]. Salus supports the standard HBase API, but its top-replicated approach provides additional robustness and performance advantages. Salus protects against memory corruption and certain

bugs in the database layer by replicating database computation as well as the data itself. Doing replication above the database level also reduces network I/O. If the database wants to reorganize data on disk (*e.g.*, via compaction), each database replica can do so on its local copy. Unfortunately, top-replicated storage is complex. The database layer must handle underlying failures as well as cooperate with other databases; in Salus, this is accomplished with a pipelined-commit protocol and Merkle trees for maintaining consistency.

Mid-bypass (Figure 3.16c) is a third option wherein the application (in this case, a database) is above the replication layer, but a well-defined API allows the application to send certain computations past the replication layer, directly to the individual replicas. Zaharia *et al.* have proposed such an abstraction for use with Spark [118]: the *RDD* (Resilient Distributed Dataset). The RDD abstraction allows the application to bypass the replication layer by sending deterministic computation directly to replicas. This gives the storage system significant flexibility over replica management. For example, rather than immediately materializing backup replicas, workers could lazily compute the backup from other replicated data, only in the event that the primary replica is lost. Similarly, the RDD abstraction could be leveraged to reduce network I/O by shipping computation directly to the data. HBase compaction could potentially be built upon two RDD transformations, *join* and *sort*, and network I/O could thus be avoided.

3.5.2 Local Compaction

We simulate the mid-bypass approach, with compaction operations shipped directly to all the replicas of compaction inputs. Figure 3.17 shows how local compaction differs from traditional compaction; network I/O is traded for local I/O, to be served by local caches or disks.

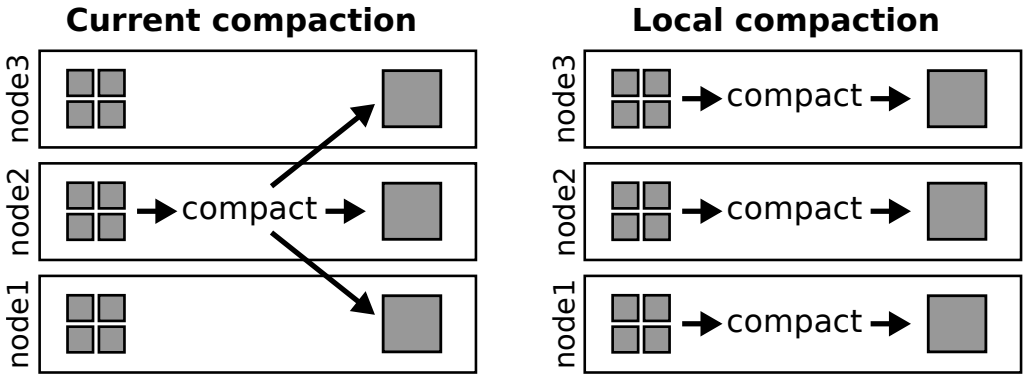

Figure 3.17: **Local-Compaction Architecture.** *The HBase architecture (left) shows how compaction currently creates a data flow with significant network I/O, represented by the two lines crossing machine boundaries. An alternative (right) shows how local reads could replace network I/O*

Figure 3.18 shows the result: a 62% reduction in network reads from 3.5 TB to 1.3 TB. The figure also shows disk reads, with and without local compaction, and with either write allocate (wa) or no-write allocate (nwa) caching policies (§3.4.1). We observe disk I/O increases sightly more than network I/O decreases. For example, with a 100 GB cache, network I/O is decreased by 2.2 GB but disk reads are increased by 2.6 GB for no-write allocate. This is unsurprising: HBase uses secondary replicas for fault tolerance rather than for reads, so secondary replicas are written once (by a flush or compaction) and read at most once (by compaction). Thus, local-compaction reads tend to (a) be misses and (b) pollute the cache with data that will not be read again. We see that write allocate still underperforms no-write allocate (§3.4.1). However, write allocate is now somewhat more competitive for large cache sizes because it is able to serve some of the data read by local compaction.

It is worth noting that local compaction requires support from the HDFS replication policy so that the compaction inputs are colocated, with implications for data-loss events as well as performance. Cidon *et al.* [22]

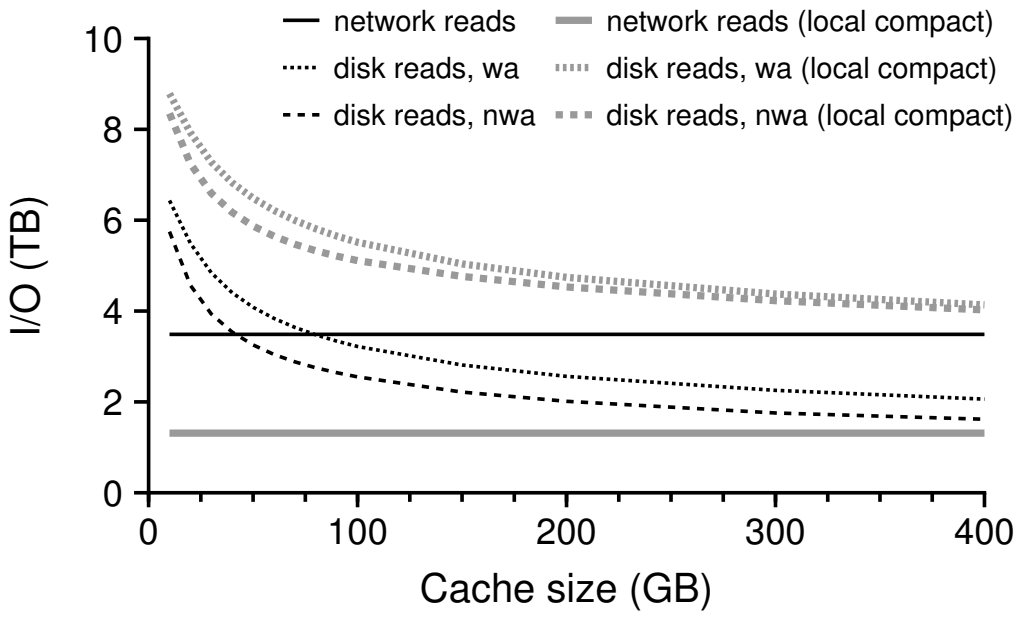

Figure 3.18: **Local-Compaction Results.** *The thick gray lines represent HBase with local compaction, and the thin black lines represent HBase currently. The solid lines represent network reads, and the dashed lines represent disk reads; long-dash represents the no-write allocate cache policy and short-dash represents write allocate.*

show that grouping inputs, as would be required for local compaction, would increase the size of data losses, but would also decrease the frequency of data losses. Thus, grouping inputs for the sake of local compaction may have either a positive or negative impact on reliability, depending on the relative costs of small and large data loss events.

Conclusion: Doing local compaction by bypassing the replication layer turns over half the network I/O into disk reads. This is a good tradeoff as network I/O is generally more expensive than sequential disk I/O.

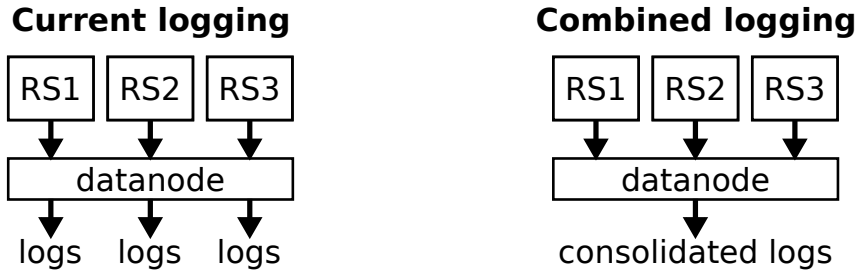

Figure 3.19: **Combined-Logging Architecture.** *Currently (left), the average DataNode will receive logs from three HBase RegionServers, and these logs will be written to different locations. An alternative approach (right) would be for HDFS to provide a special logging API which allows all the logs to be combined so that disk seeks are reduced.*

3.5.3 Combined Logging

We now consider the interaction between replication and HBase logging. Figure 3.19 shows how (currently) a typical DataNode will receive log writes from three RegionServers (because each RegionServer replicates its logs to three DataNodes). These logs are currently written to three different local files, causing seeks. Such seeking could be reduced if HDFS were to expose a special logging feature that merges all logical logs into a single physical log on a dedicated disk as illustrated.

We simulate combined logging and measure performance for requests which go to disk; we consider latencies for foreground reads (defined in Section 3.4.1), compaction, and logging. Figure 3.20 reports the results for varying numbers of disks. The latency of log writes decreases dramatically with combined logging; for example, with 15 disks, the latency is decreased by a factor of six. Compaction requests also experience modest gains due to less competition for disk seeks. Currently, neither logging nor compaction block the end user, so we also consider the performance of foreground reads. For this metric, the gains are small, *e.g.*, latency only decreases by 3.4% with 15 disks. With just 10 disks, dedicating one disk to logging slightly hurts user reads.

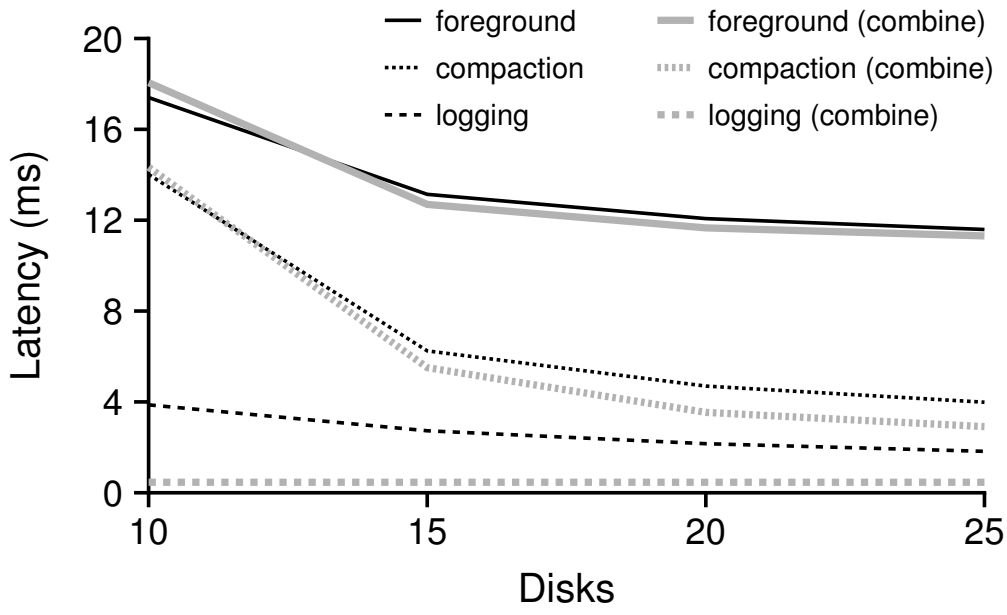

Figure 3.20: **Combined Logging Results.** *Disk latencies for various activities are shown, with (gray) and without (black) combined logging.*

Conclusion: Merging multiple HBase logs on a dedicated disk reduces logging latencies by a factor of 6. However, `put` requests do not currently block until data is flushed to disks, and the performance impact on foreground reads is negligible. Thus, the additional complexity of combined logging is likely not worthwhile given the current durability guarantees. However, combined logging could enable HBase, at little performance cost, to give the additional guarantee that data is on disk before a `put` returns. Providing such a guarantee would make logging a foreground activity.

3.6 Summary

We have presented a detailed multilayer study of storage I/O for Facebook Messages. Our combined approach of analysis and simulation allowed us to identify potentially useful changes and then evaluate those changes. We have four major conclusions.

First, the special handling received by writes make them surprisingly expensive. At the HDFS level, the read/write ratio is 99/1, excluding HBase compaction and logging overheads. At the disk level, the ratio is write-dominated at 36/64. Logging, compaction, replication, and caching all combine to produce this write blowup. Thus, optimizing writes is very important even for especially read-heavy workloads such as FM.

Second, the GFS-style architecture is based on workload assumptions such as "high sustained bandwidth is more important than low latency" [40]. For FM, many of these assumptions no longer hold. For example, we demonstrate (§3.4.1) just the opposite is true for FM: because I/O is highly random, bandwidth matters little, but latency is crucial. Similarly, files were assumed to be very large, in the hundreds or thousands of megabytes. This traditional workload implies a high data-to-metadata ratio, justifying the one-NameNode design of GFS and HDFS. By contrast, FM is dominated by small files; perhaps the single-NameNode design should be revisited.

Third, FM storage is built upon layers of independent subsystems. This architecture has the benefit of simplicity; for example, because HBase stores data in a replicated store, it can focus on high-level database logic instead of dealing with dying disks and other types of failure. Layering is also known to improve reliability, *e.g.*, Dijkstra found layering "proved to be vital for the verification and logical soundness" of an OS [29]. Unfortunately, we find that the benefits of simple layering are not free. In particular, we showed (§3.5) that building a database over a replication layer causes additional network I/O and increases workload randomness

at the disk layer. Fortunately, simple mechanisms for sometimes bypassing replication can reduce layering costs.

Fourth, the cost of flash has fallen greatly, prompting Gray's proclamation that "tape is dead, disk is tape, flash is disk" [43]. To the contrary, we find that for FM, flash is not a suitable replacement for disk. In particular, the cold data is too large to fit well in flash (§3.3.1) and the hot data is too large to fit well in RAM (§3.4.1). However, our evaluations show that architectures with a small flash tier have a positive cost/performance tradeoff compared to systems built on disk and RAM alone.

4
Docker Measurement

Containers are widely used to isolate applications, and their popularity is rapidly growing [63, 74, 108]. Cloud compute has long been based on the virtualization of hardware, but using virtual machines is relatively expensive (*e.g.*, each application must have its own operating system). Like virtual machines, containers provide isolation by virtualizing resources, but at a higher level. For example, a VMM virtualizes network cards, but a container virtualizes ports. With containers, many applications can share the same operating system because the resources virtualized by containers are the high-level resources exposed by an operating system.

Container-like support not a new idea [84], but serious support for containers in Linux is fairly recent [25]. This new support has influenced how people build applications: isolation is cheaper, so there is more motivation to split applications into smaller components, or microservices [24]. Microservices can be organized in a variety of ways, *e.g.*, layers, trees, or graphs, but in all these patterns, each microservice will have significant control over its own execution environment. In this chapter, we explore the question: *what is the cost of deploying and providing storage for these many different microservice environments?*

In order to focus our study, we analyze one specific container deployment tool, Docker [70]. For our workload, we execute 57 applications running inside Docker containers. Unless an application requires specific kernel versions or extensions, the application is runnable in a variety of

environments: on bare metal, in a virtual machine, or in a Docker container. Many measurements for these applications will likely be similar regardless of the environment. However, our goal is to specifically understand Docker patterns, so we bias our analysis towards conditions under which measurements will tend to be unique to Docker. In particular, this means we focus more on application startup and less on long-running performance characteristics. Startup is a known problem with containers [108], and the representation of applications as compressed layers (as opposed to, say, virtual disk images as with virtual machines) leads to new I/O patterns.

We first give a brief background on Docker (§4.1) and describe our analysis methodology based on the HelloBench workloads (§4.2). Next, we present our findings regarding image data (§4.3), distribution performance (§4.4), layering properties (§4.5), and I/O determinism (§4.6). Finally, we summarize our findings (§4.7).

4.1 Background

We now describe Docker's framework (§4.1.1), storage interface (§4.1.2), and default storage driver (§4.1.3).

4.1.1 Version Control for Containers

While Linux has always used virtualization to isolate memory, cgroups [25] (Linux's container implementation) virtualizes a broader range of resources by providing six new namespaces, for file-system mount points, IPC queues, networking, host names, process IDs, and user IDs [73]. Linux cgroups were first released in 2007, but widespread container use is a more recent phenomenon, coinciding with the availability of new container management tools such as Docker (released in 2013). With Docker, a single command such as "`docker run -it ubuntu bash`" will pull Ubuntu pack-

ages from the Internet, initialize a file system with a fresh Ubuntu installation, perform the necessary cgroup setup, and return an interactive bash session in the environment.

This example command has several parts. First, "ubuntu" is the name of an *image*. Images are read-only copies of file-system data, and typically contain application binaries, a Linux distribution, and other packages needed by the application. Bundling applications as Docker images is convenient because the distributor can select a specific set of packages (and their versions) that will be used wherever the application is run. Second, "run" is an operation to perform on an image; the run operation creates an initialized root file system based on the image to use for a new *container*. Other operations include "push" (for publishing new images) and "pull" (for fetching published images from a central location); an image is automatically pulled if the user attempts to run a non-local image. Third, "bash" is the program to start within the container; the user may specify any executable in the given image.

Docker manages image data much the same way traditional version-control systems manage code. This model is suitable for two reasons. First, there may be different branches of the same image (*e.g.*, "ubuntu:latest" or "ubuntu:12.04"). Second, images naturally build upon one another. For example, the Ruby-on-Rails image builds on the Rails image, which in turn builds on the Debian image. Each of these images represent a new *commit* over a previous commit; there may be additional commits that are not tagged as runnable images. When a container executes, it starts from a committed image, but files may be modified; in version-control parlance, these modifications are referred to as unstaged changes. The Docker "commit" operation turns a container and its modifications into a new read-only image. In Docker, a *layer* refers to either the data of a commit or to the unstaged changes of a container.

Docker worker machines run a local Docker *daemon*. New containers

and images may be created on a specific worker by sending commands to its local daemon. Image sharing is accomplished via centralized *registries* that typically run on machines in the same cluster as the Docker workers. Images may be published with a push from a daemon to a registry, and images may be deployed by executing pulls on a number of daemons in the cluster. Only the layers not already available on the receiving end are transferred. Layers are represented as gzip-compressed tar files over the network and on the registry machines. Representation on daemon machines is determined by a pluggable storage driver.

4.1.2 Storage Driver Interface

Docker containers access storage in two ways. First, users may mount directories on the host within a container. For example, a user running a containerized compiler may mount her source directory within the container so that the compiler can read the code files and produce binaries in the host directory. Second, containers need access to the Docker layers used to represent the application binaries and libraries. Docker presents a view of this application data via a mount point that the container uses as its root file system. Container storage and mounting is managed by a Docker storage driver; different drivers may choose to represent layer data in different ways. The methods a driver must implement are shown in Table 4.1 (some uninteresting functions and arguments are not shown). All the functions take a string "id" argument that identifies the layer being manipulated.

The `Get` function requests that the driver mount the layer and return a path to the mount point. The mount point returned should contain a view of not only the "id" layer, but of all its ancestors (*e.g.*, files in the parent layer of the "id" layer should be seen during a directory walk of the mount point). `Put` unmounts a layer. `Create` copies from a parent layer to create a new layer. If the parent is `NULL`, the new layer should be empty.

Method	Description
Get(id)=dir	mount "id" layer file system, return mount point
Put(id)	unmount "id" layer file system
Create(parent, id)	logically copy "parent" layer to "id" layer
Diff(parent, id)=tar	return compressed tar of changes in "id" layer
ApplyDiff(id, tar)	apply changes in tar to "id" layer

Table 4.1: **Docker Driver API.**

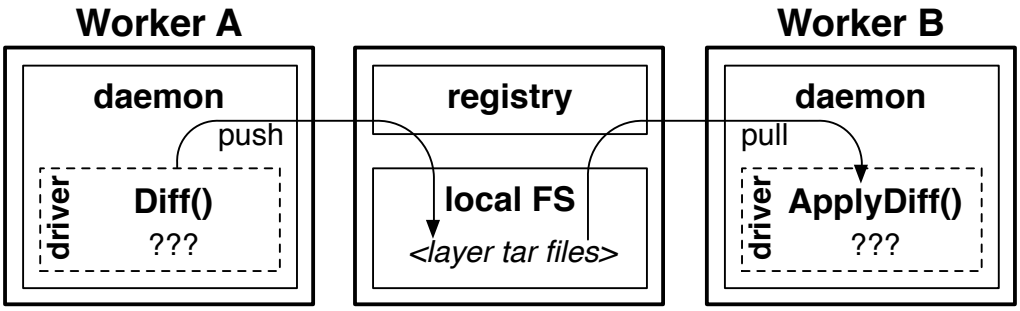

Figure 4.1: **Diff and ApplyDiff.** *Worker A is using `Diff` to package local layers as compressed tars for a push. B is using `ApplyDiff` to convert the tars back to the local format. Local representation varies depending on the driver, as indicated by the question marks.*

Docker calls `Create` to (1) provision file systems for new containers, and (2) allocate layers to store data from a pull.

`Diff` and `ApplyDiff` are used during Docker push and pull operations respectively, as shown in Figure 4.1. When Docker is pushing a layer, `Diff` converts the layer from the local representation to a compressed tar file containing the files of the layer. `ApplyDiff` does the opposite: given a tar file and a local layer it decompresses the tar file over the existing layer.

Figure 4.2 shows the driver calls that are made when a four-layer image (*e.g.*, ubuntu) is run for the first time. Four layers are created during the image pull; two more are created for the container itself. Layers A-D represent the image. The `Create` for A takes a `NULL` parent, so A is ini-

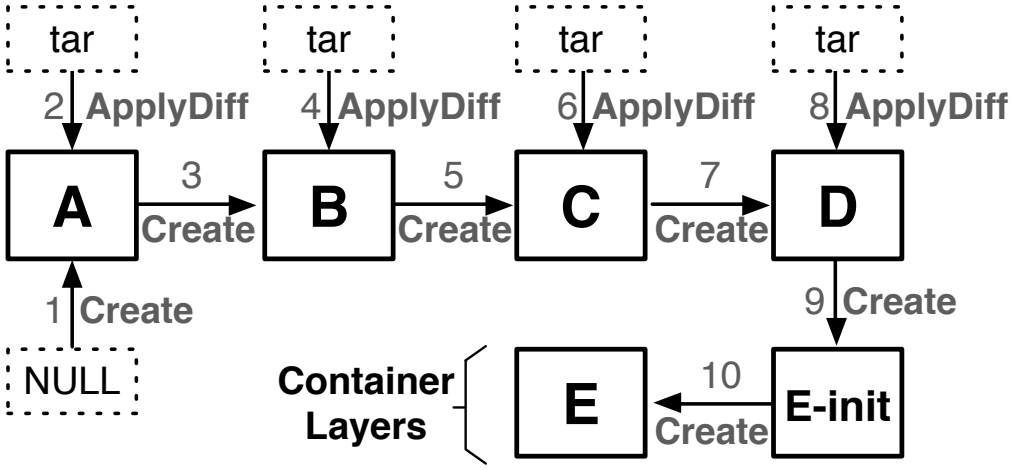

Figure 4.2: **Cold Run Example.** *The driver calls that are made when a four-layer image is pulled and run are shown. Each arrow represents a call (`Create` or `ApplyDiff`), and the nodes to which an arrow connects indicate arguments to the call. Thick-bordered boxes represent layers. Integers indicate the order in which functions are called.*

tially empty. The subsequent `ApplyDiff` call, however, tells the driver to add the files from the pulled tar to A. Layers B-D are are each populated with two steps: a copy from the parent (via `Create`), and the addition of files from the tar (via `ApplyDiff`). After step 8, the pull is complete, and Docker is ready to create a container. It first creates a read-only layer E-init, to which it adds a few small initialization files, and then it creates E, the file system the container will use as its root.

4.1.3 AUFS Driver Implementation

The AUFS storage driver is a common default for Docker distributions. This driver is based on the AUFS file system (Another Union File System). Union file systems do not store data directly on disk, but rather use another file system (*e.g.*, ext4) as underlying storage.

A union mount point provides a view of multiple directories in the un-

derlying file system. AUFS is mounted with a list of directory paths in the underlying file system. During path resolution, AUFS iterates through the list of directories; the first directory to contain the path being resolved is chosen, and the inode from that directory is used. AUFS also supports special *whiteout* files to make it appear that certain files in lower layers have been deleted; this technique is analogous to deletion markers in other layered systems (*e.g.*, LSM databases [76]). AUFS also supports COW (copy-on-write) at file granularity; upon write, files in lower layers are copied to the top layer before the write is allowed to proceed.

The AUFS driver takes advantage the AUFS file system's layering and copy-on-write capabilities while also accessing the file system underlying AUFS directly. The driver creates a new directory in the underlying file system for each layer it stores. An `ApplyDiff` simple untars the archived files into the layer's directory. Upon a `Get` call, the driver uses AUFS to create a unioned view of a layer and its ancestors. The driver uses AUFS's COW to efficiently copy layer data when `Create` is called. Unfortunately, as we will see, COW at file granularity has some performance problems (§4.5).

4.2 Measurement Methodology

In order to study container startup, we choose a set of applications to run inside Docker containers, and write scripts to pull and launch these containers in a controlled and repeatable way. We call our suite of applications HelloBench. HelloBench directly executes Docker commands, so pushes, pulls, and runs can be measured independently.

Although most applications could be run inside containers, we choose all our applications from the public Docker Hub library [30]. Our reasoning is that this set of applications will be more frequently used in containers.

Linux Distro:	alpine, busybox, centos, cirros, crux, debian, fedora, mageia, opensuse, oraclelinux, ubuntu, ubuntu-debootstrap, ubuntu-upstart
Database:	cassandra, crate, elasticsearch, mariadb, mongo, mysql, percona, postgres, redis, rethinkdb
Language:	clojure, gcc, golang, haskell, hylang, java, jruby, julia, mono, perl, php, pypy, python, r-base, rakudo-star, ruby, thrift
Web Server:	glassfish, httpd, jetty, nginx, php-zendserver, tomcat
Web Framework:	django, iojs, node, rails
Other:	drupal, ghost, hello-world, jenkins, rabbitmq, registry, sonarqube

Table 4.2: **HelloBench Workloads.** *HelloBench runs 57 different container images pulled from the Docker Hub.*

As of June 1, 2015, there were 72 containerized applications on Docker Hub. We evaluated all of these for inclusion in HelloBench, and selected applications that were runnable with minimal configuration and do not depend on other containers. For example, WordPress is not included because a WordPress container depends on a separate MySQL container. The final application suite contains 57 Docker container images.

Table 4.2 lists the images used by HelloBench. We divide the images into six broad categories as shown. Some classifications are somewhat subjective; for example, the Django image contains a web server, but most would probably consider it a web framework.

The HelloBench harness measures startup time by either running the simplest possible task in the container or waiting until the container reports readiness. For the language containers, the task typically involves compiling or interpreting a simple "hello world" program in the applicable language. The Linux distro images execute a very simple shell com-

mand, typically "`echo hello`". For long-running servers (particularly databases and web servers), HelloBench measures the time until the container writes an "up and ready" (or similar) message to standard out. For particularly quiet servers, an exposed port is polled until there is a response.

HelloBench images each consist of many layers, some of which are shared between containers. Figure 4.3 shows the relationships between layers. Across the 57 images, there are 550 nodes and 19 roots. In some cases, a tagged image serves as a base for other tagged images (*e.g.*, "ruby" is a base for "rails"). Only one image consists of a single layer: "alpine", a particularly lightweight Linux distribution. Application images are often based on non-latest Linux distribution images (*e.g.*, older versions of Debian); that is why multiple images will often share a common base that is not a solid black circle.

In order to evaluate how representative HelloBench is of commonly used images, we counted the number of pulls to every Docker Hub library image [30] on January 15, 2016 (7 months after the original HelloBench images were pulled). During this time, the library grew from 72 to 94 images. Figure 4.4 shows pull counts to the 94 images, broken down by HelloBench category. HelloBench is representative of popular images, accounting for 86% of all pulls. Most pulls are to Linux distribution bases (*e.g.*, BusyBox and Ubuntu). Databases (*e.g.*, Redis and MySQL) and web servers (*e.g.*, nginx) are also popular.

We use HelloBench throughout the remainder of our analysis for collecting traces and measuring performance. All performance measurements are taken from a virtual machine running on an PowerEdge R720 host with 2 GHz Xeon CPUs (E5-2620). The VM is provided 8 GB of RAM, 4 CPU cores, and a virtual disk backed by a Tintri T620 [105]. The server and VMstore had no other load during the experiments.

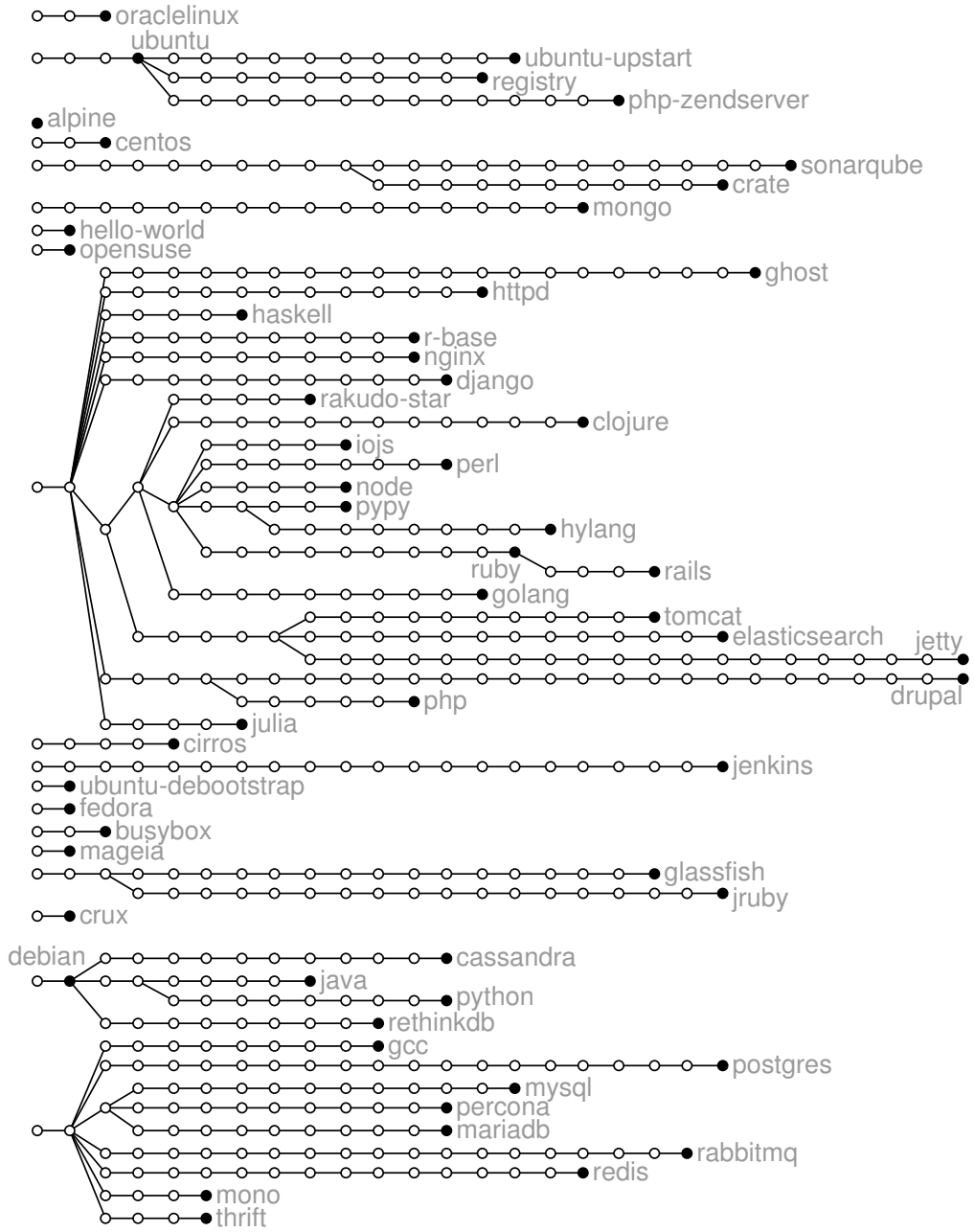

Figure 4.3: **HelloBench Hierarchy.** *Each circle represents a layer. Filled circles represent layers tagged as runnable images. Deeper layers are to the left.*

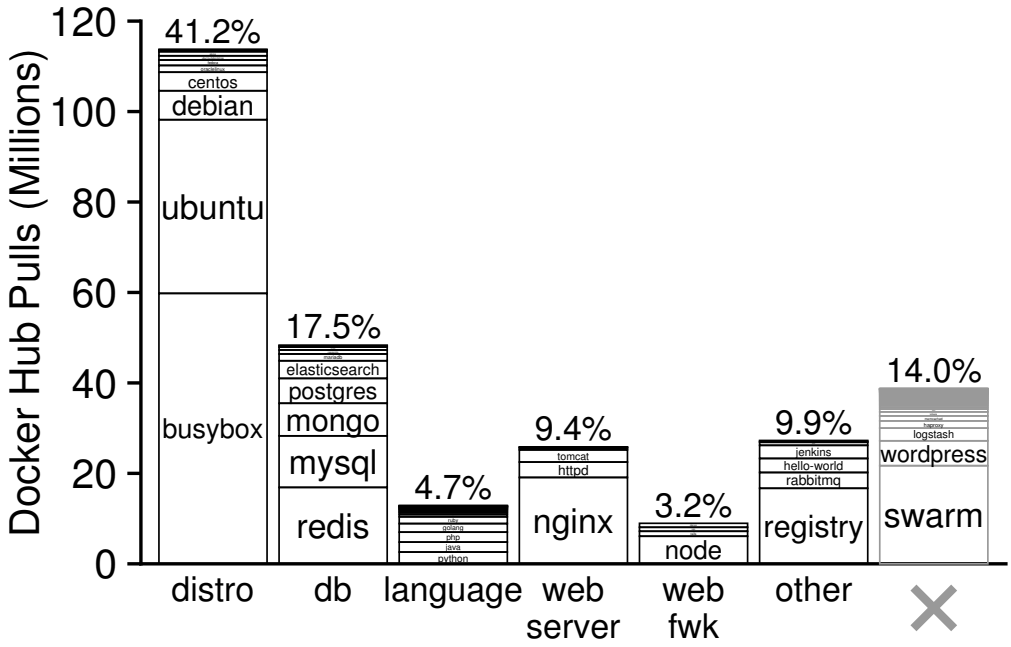

Figure 4.4: **Docker Hub Pulls.** *Each bar represents the number of pulls to the Docker Hub library, broken down by category and image. The far-right gray bar represents pulls to images in the library that are not run by HelloBench.*

4.3 Image Data

We begin our analysis by studying the HelloBench images pulled from the Docker Hub. For each image, we take three measurements: its compressed size, uncompressed size, and the number of bytes read from the image when HelloBench executes. We measure reads by running the workloads over a block device traced with `blktrace` [9]. Figure 4.5 shows a CDF of these three numbers. We observe that only 20 MB of data is read on median, but the median image is 117 MB compressed and 329 MB uncompressed.

We break down the read and size numbers by category in Figure 4.6. The largest relative waste is for distro workloads (30× and 85× for compressed and uncompressed respectively), but the absolute waste is also

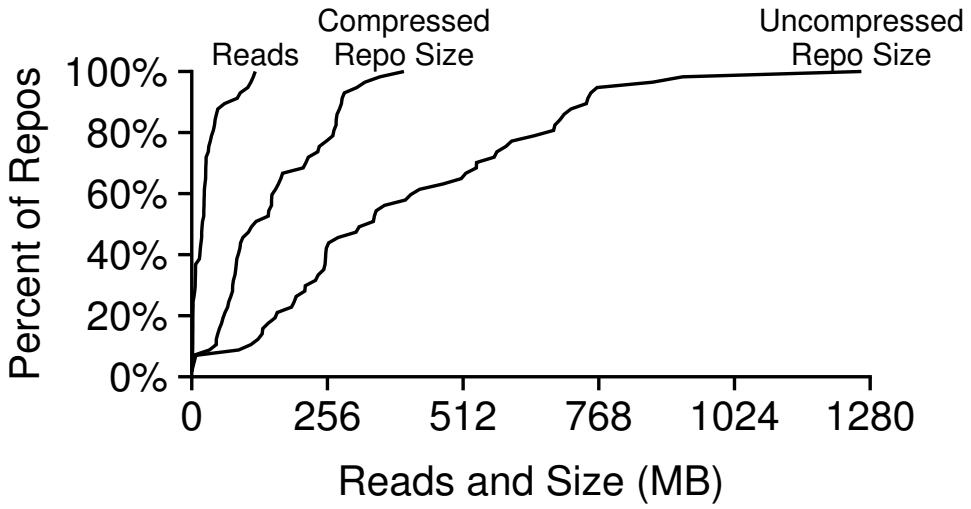

Figure 4.5: **Data Sizes (CDF).** *Distributions are shown for the number of reads in the HelloBench workloads and for the uncompressed and compressed sizes of the HelloBench images.*

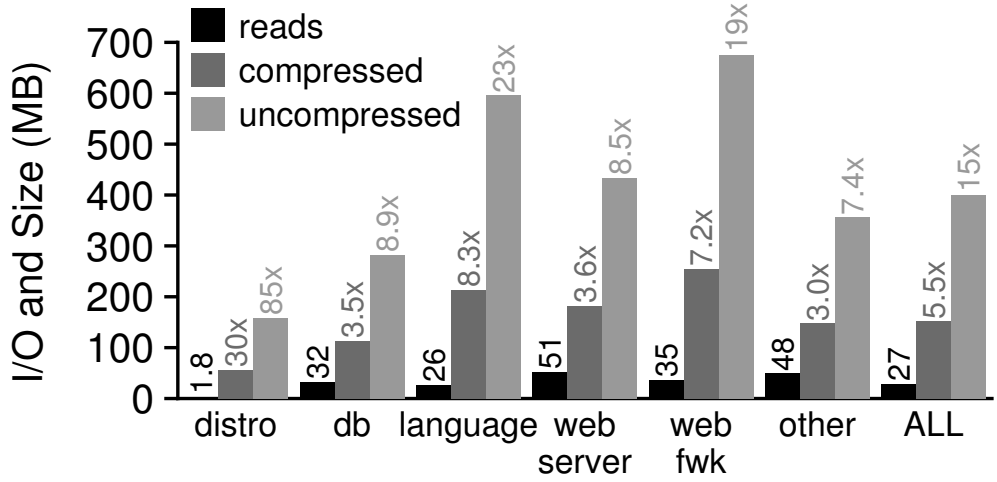

Figure 4.6: **Data Sizes (By Category).** *Averages are shown for each category. The size bars are labeled with amplification factors, indicating the amount of transferred data relative to the amount of useful data (i.e., the data read).*

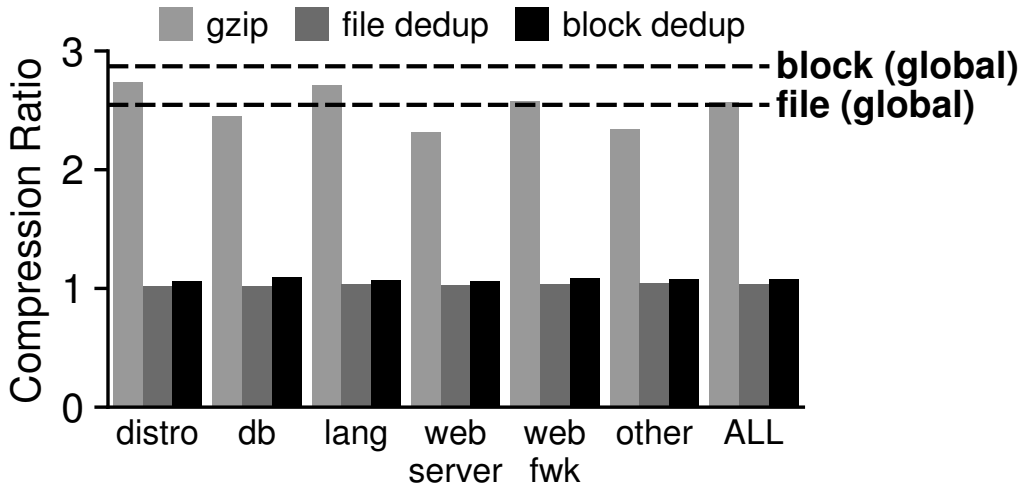

Figure 4.7: **Compression and Deduplication Rates.** *The y-axis represents the ratio of the size of the raw data to the size of the compressed or deduplicated data. The bars represent per-image rates. The lines represent rates of global deduplication across the set of all images.*

smallest for this category. Absolute waste is highest for the language and web framework categories. Across all images, only 27 MB is read on average; the average uncompressed image is 15× larger, indicating only 6.4% of image data is needed for container startup.

Although Docker images are much smaller when compressed as gzip archives, this format is not suitable for running containers that need to modify data. Thus, workers typically store data uncompressed, which means that compression reduces network I/O but not disk I/O. Deduplication is a simple alternative to compression that is suitable for updates. We scan HelloBench images for redundancy between blocks of files to compute the effectiveness of deduplication. Figure 4.7 compares gzip compression rates to deduplication, at both file and block (4 KB) granularity. Bars represent rates over single images. Whereas gzip achieves rates between 2.3 and 2.7, deduplication does poorly on a per-image basis. Deduplication across all images, however, yields rates of 2.6 (file granu-

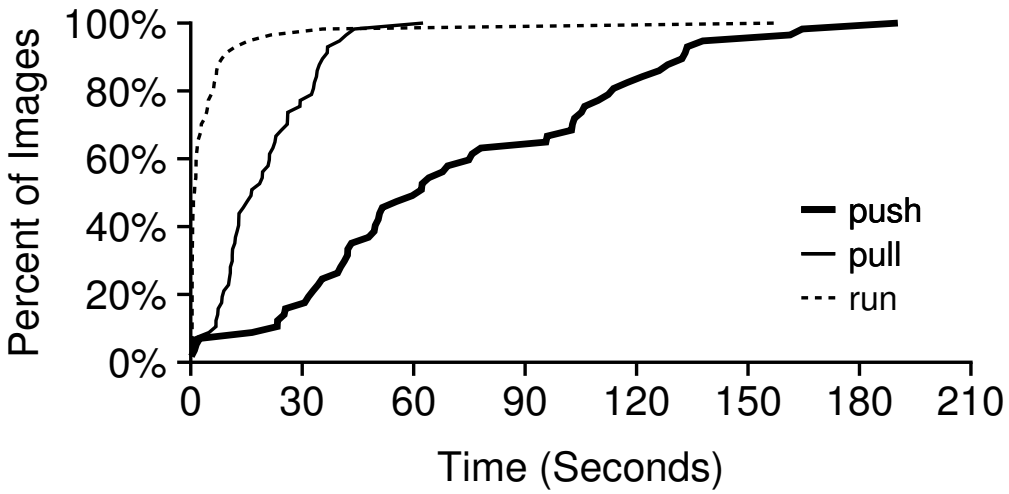

Figure 4.8: **Operation Performance (CDF).** *A distribution of push, pull, and run times for HelloBench are shown for Docker with the AUFS storage driver.*

larity) and 2.8 (block granularity).

Conclusion: the amount of data read during execution is much smaller than the total image size, either compressed or uncompressed. Image data is sent over the network compressed, then read and written to local storage uncompressed, so overheads are high for both network and disk. One way to decrease overheads would be to build leaner images with fewer installed packages. Alternatively, image data could be lazily pulled as a container needs it. We also saw that global block-based deduplication is an efficient way to represent image data, even compared to gzip compression.

4.4 Operation Performance

Once built, containerized applications are often deployed as follows: the developer *pushes* the application image once to a central registry, a number of workers *pull* the image, and each worker *runs* the application. We

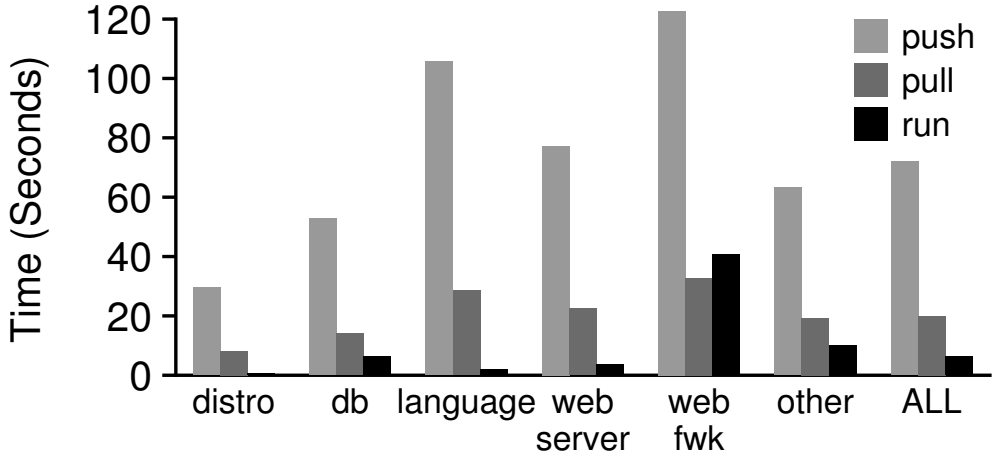

Figure 4.9: **Operation Performance (By Category).** *Averages are shown for each category.*

measure the latency of these operations with HelloBench, reporting CDFs in Figure 4.8. Median times for push, pull, and run are 61, 16, and 0.97 seconds respectively.

Figure 4.9 breaks down operation times by workload category. The pattern holds in general: runs are fast while pushes and pulls are slow. Runs are fastest for the distro and language categories (0.36 and 1.9 seconds respectively). The average times for push, pull, and run are 72, 20, and 6.1 seconds respectively. Thus, 76% of startup time will be spent on pull when starting a new image hosted on a remote registry.

As pushes and pulls are slowest, we want to know whether these operations are merely high latency, or whether they are also costly in a way that limits throughput even if multiple operations run concurrently. To study scalability, we concurrently push and pull varying numbers of artificial images of varying sizes. Each image contains a single randomly generated file. We use artificial images rather than HelloBench images in order to create different equally-sized images. Figure 4.10 shows that the total time scales roughly linearly with the number of images and im-

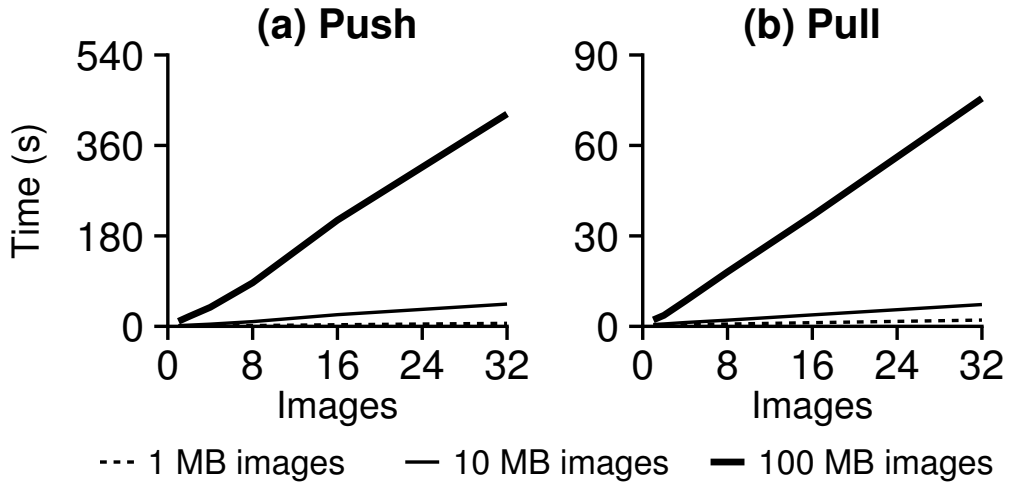

Figure 4.10: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

age size. Thus, pushes and pulls are not only high-latency, they consume network and disk resources, limiting scalability.

Conclusion: container startup time is dominated by pulls; 76% of the time spent on a new deployment will be spent on the pull. Publishing images with push will be painfully slow for programmers who are iteratively developing their application, though this is likely a less frequent case than multi-deployment of an already published image. Most push work is done by the storage driver's `Diff` function, and most pull work is done by the `ApplyDiff` function (§4.1.2). Optimizing these driver functions would improve distribution performance.

4.5 Layers

Image data is typically split across a number of layers. The AUFS driver composes the layers of an image at runtime to provide a container a complete view of the file system. In this section, we study the performance

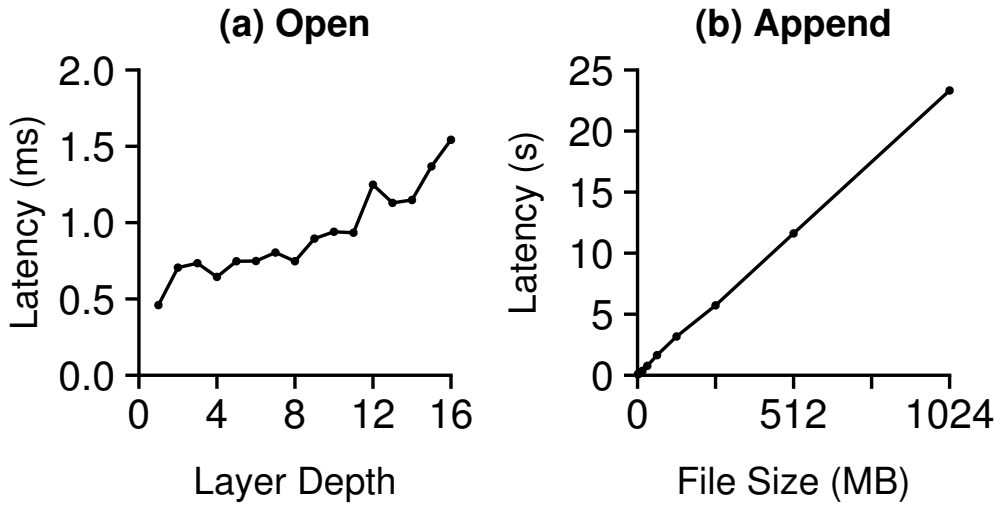

Figure 4.11: **AUFS Performance.** *Left: the latency of the* `open` *system call is shown as a function of the layer depth of the file. Right: the latency of a one-byte append is shown as a function of the size of the file that receives the write.*

implications of layering and the distribution of data across layers. We start by looking at two performance problems (Figure 4.11) to which layered file systems are prone: lookups to deep layers and small writes to non-top layers.

First, we create (and compose with AUFS) 16 layers, each containing 1K empty files. Then, with a cold cache, we randomly open 10 files from each layer, measuring the open latency. Figure 4.11a shows the result (an average over 100 runs): there is a strong correlation between layer depth and latency. Second, we create two layers, the bottom of which contains large files of varying sizes. We measure the latency of appending one byte to a file stored in the bottom layer. As shown by Figure 4.11b, the latency of small writes correspond to the file size (not the write size), as AUFS does COW at file granularity. Before a file is modified, it is copied to the topmost layer, so writing one byte can take over 20 seconds. Fortunately, small writes to lower layers induce a one-time cost per container; subsequent writes will be faster because the large file will have been copied to

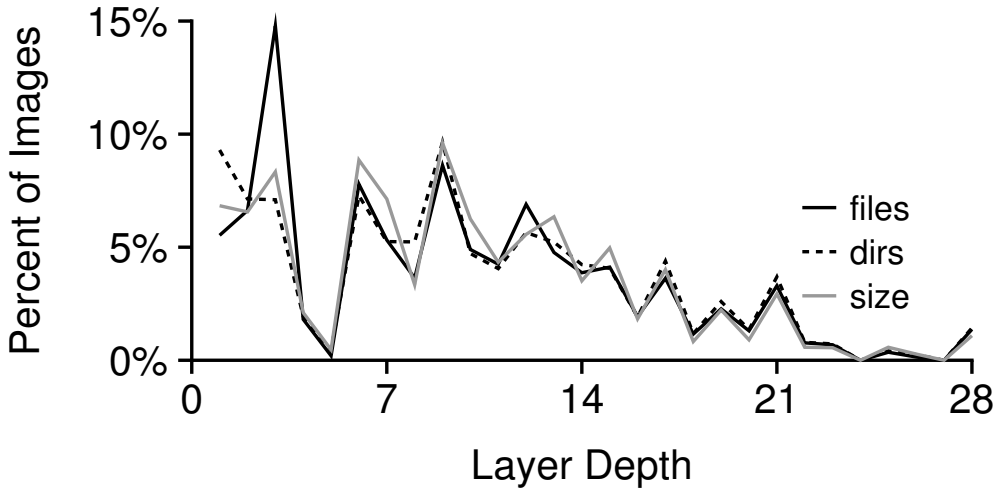

Figure 4.12: **Data Depth.** *The lines show mass distribution of data across image layers in terms of number of files, number of directories, and bytes of data in files.*

the top layer.

Having considered how layer depth corresponds with performance, we now ask, *how deep is data typically stored for the HelloBench images?* Figure 4.12 shows the percentage of total data (in terms of number of files, number of directories, and size in bytes) at each depth level. The three metrics roughly correspond. Some data is as deep as level 28, but mass is more concentrated to the left. Over half the bytes are at depth of at least nine.

We now consider the variance in how data is distributed across layers, measuring, for each image, what portion (in terms of bytes) is stored in the topmost layer, bottommost layer, and whatever layer is largest. Figure 4.13 shows the distribution: for 79% of images, the topmost layer contains 0% of the image data. In contrast, 27% of the data resides in the bottommost layer in the median case. A majority of the data typically resides in a single layer.

Conclusion: for layered file systems, data stored in deeper layers is

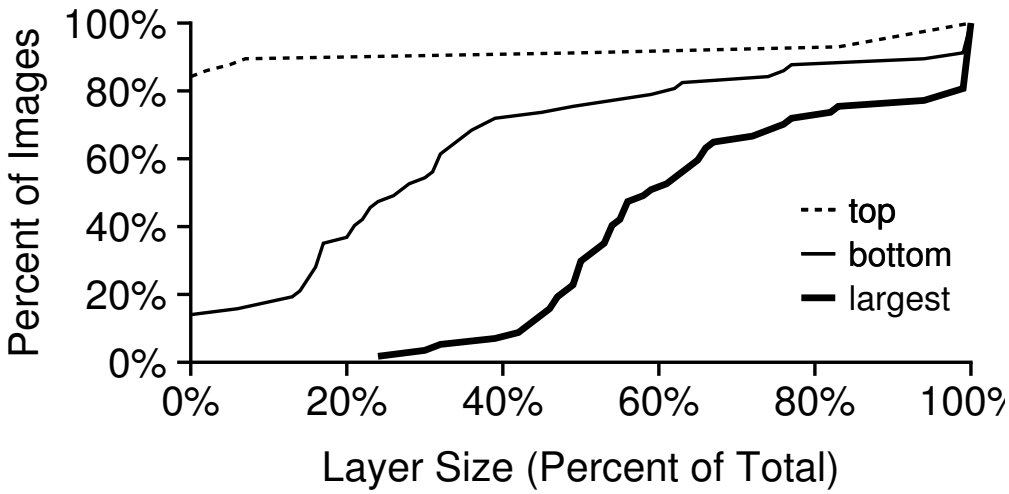

Figure 4.13: **Layer Size (CDF).** *The size of a given layer is measured relative to the total image size (x-axis), and the distribution of relative sizes is shown. The plot considers the topmost layer, bottommost layer, and whichever layer happens to be largest. All measurements are in terms of file bytes.*

slower to access. Unfortunately, Docker images tend to be deep, with at least half of file data at depth nine or greater. Flattening layers is one technique to avoid these performance problems; however, flattening could potentially require additional copying and void the other COW benefits that layered file systems provide.

4.6 Caching

We now consider the case where the same worker runs the same image more than once. In particular, we want to know whether I/O from the first execution can be used to prepopulate a cache to avoid I/O on subsequent runs. Towards this end, we run every HelloBench workload twice consecutively, collecting block traces each time. We compute the portion of reads during the second run that could potentially benefit from cache state populated by reads during the first run.

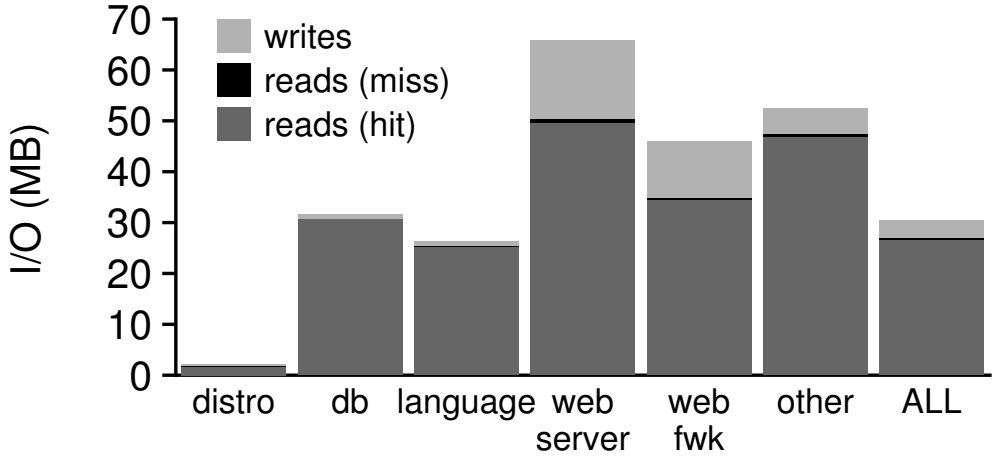

Figure 4.14: **Repeated I/O.** *The bars represent total I/O done for the average container workload in each category. Bar sections indicate read/write ratios. Reads that could have potentially been serviced by a cache populated by previous container execution are dark gray.*

Figure 4.14 shows the reads and writes for the second run. Reads are broken into hits and misses. For a given block, only the first read is counted (we want to study the workload itself, not the characteristics of the specific cache beneath which we collected the traces). Across all workloads, the read/write ratio is 88/12. For distro, database, and language workloads, the workload consists almost completely of reads. Of the reads, 99% could potentially be serviced by cached data from previous runs.

Conclusion: The same data is often read during different runs of the same image, suggesting cache sharing will be useful when the same image is executed on the same machine many times. In large clusters with many containerized applications, repeated executions will be unlikely unless container placement is highly restricted. Also, other goals (*e.g.*, load balancing and fault isolation) may make colocation uncommon. However, repeated executions are likely common for containerized utility programs (*e.g.*, python or gcc) and for applications running in small clusters.

Our results suggest these latter scenarios would benefit from cache sharing.

4.7 Summary

In this chapter, we have studied 57 containerized applications and startup applications, representing 86% of the images pulled from the Docker Hub. Our analysis leads to several findings, with strong implications for how to build container storage.

Some of our key findings about Docker resemble findings in a measurement study on Google's Borg container deployment platform [108]. In Borg, 80% of startup time is spent on package installation; in Docker, 76% of startup time is spent pulling image data. We find that 94% of the data installed during pull is not actually used during container startup, suggesting image data should be pulled lazily.

We also make several new observations regarding Docker's on-disk representation of image data. Docker layers are composed to provide a single view of all file data, but we show accesses to data in lower layers are slow with Docker's AUFS storage driver. Unfortunately, most data is deep; over half the bytes are at a layer depth of at least nine.

In Chapter 5, we discuss further design implications gleaned from our analysis, and describe Slacker, a new Docker storage driver that is based on our findings.

5

Slacker: A Lazy Docker Storage Driver

In Chapter 4, we described our measurements and analysis of Docker I/O upon the launch of a variety of applications. In this chapter, we describe the design and implementation of a new Docker storage driver that is driven by our analysis. Our new driver is called Slacker because it lazily copies and fetches container data. Our analysis showed that the vast majority of the work vanilla Docker does is unnecessary, so Slacker avoids doing any such work until necessary. Slacker is built with five goals:

1. **Make pushes and pulls very fast.** Our Docker analysis and other analysis of Google Borg suggest distribution is a primary problem for containers.

2. **Introduce no slowdown for long-running containers.** Long-running applications are still a common use case, so a solution that creates a difficult tradeoff between startup latency and steady-state performance would greatly reduce the solution's generality.

3. **Reuse existing storage systems whenever possible.** Our analysis suggests that data should be lazily copied and fetched. Copy-on-write and lazy propagation are not new strategies in systems, so we seek to reuse rather than reimplement.

4. **Make no changes to the Docker registry or daemon except in the storage-driver plugin.** Docker bases storage on a general API, en-

abling flexible solutions. For example, when the underlying file system (*e.g.*, btrfs) provides copy-on-write functionality, Docker can use those features directly, without needing to rely on an inefficient layered file system (*e.g.*, AUFS). By restricting ourselves to using Docker's existing framework, we can evaluate the generality of the API and suggest improvements.

5. **Utilize the powerful primitives provided by a modern storage server.** The Docker btrfs driver utilizes the copy-on-write functionality of a local file system to speed up local execution. We similarly hope to build a driver that utilizes the copy-on-write functionality of a network file system to speed up distribution. Specifically, we use the functionality exposed by a Tintri VMstore [106].

We begin our discussion by considering the design implications from our measurement study in more detail (§5.1) and giving an overview of the Slacker architecture (§5.2). We then describe how Slacker represents Docker layers (§5.3), Slacker's integration with VMstore (§5.4), optimizations for snapshot and clone (§5.5), and kernel modifications that Slacker uses (§5.6). We next evaluate Slacker's performance (§5.7) and usefulness (§5.8). Finally, we conclude by discussing potential modifications to the Docker framework itself (§5.9) and summarizing our work (§5.10).

5.1 Measurement Implications

In this section, we recap the findings of Chapter 4 and discuss the implications for the design of a new Docker storage driver.

In Section 4.4, we measure push, pull, and run performance for HelloBench workloads running on Docker with AUFS. While runs are fast, pushes and pulls are very slow. For example, 76% of the time spent on a new deployment will be spent on the pull. Publishing images with push will

be also painfully slow for programmers who are iteratively developing their application, though this is likely a less frequent case than multi-deployment of an already published image. Inside Docker, most push work is done by the storage driver's `Diff` function, and most pull work is done by the `ApplyDiff` function (§4.1.2). Thus, optimizing these driver functions would improve distribution performance.

Section 4.3 provides more insight into why `Diff` and `ApplyDiff` are slow relative to the run operation in Docker. Both functions copy image data over the network compressed and to disk uncompressed, and the average image size is 150 MB compressed and 399 MB uncompressed. In contrast, only 27 MB of data is needed during container startup upon run. Thus, `Diff` and `ApplyDiff` cause significant I/O that is not necessary for startup. One way to decrease overheads would be to build leaner images with fewer installed packages. Alternatively, image data could be lazily pulled as a container needs it. We also saw that global block-based deduplication is an efficient way to represent image data, even compared to gzip compression.

In Section 4.5, we show that for layered file systems, data stored in deeper layers is slower to access. Unfortunately, Docker images tend to be deep, with at least half of file data at depth nine or greater. Flattening layers is one technique to avoid these performance problems; however, flattening could potentially require additional copying and void the other COW benefits that layered file systems provide. One solution would be to retain layers, but use a more efficient COW mechanism. For example, block-level traversals are faster than file-level traversals.

Section 4.6 measures how similar reads are across multiple runs of the same application. For HelloBench, the same data is often read during different runs of the same image, suggesting cache sharing will be useful when the same image is executed on the same machine many times. In large clusters with many containerized applications, repeated executions

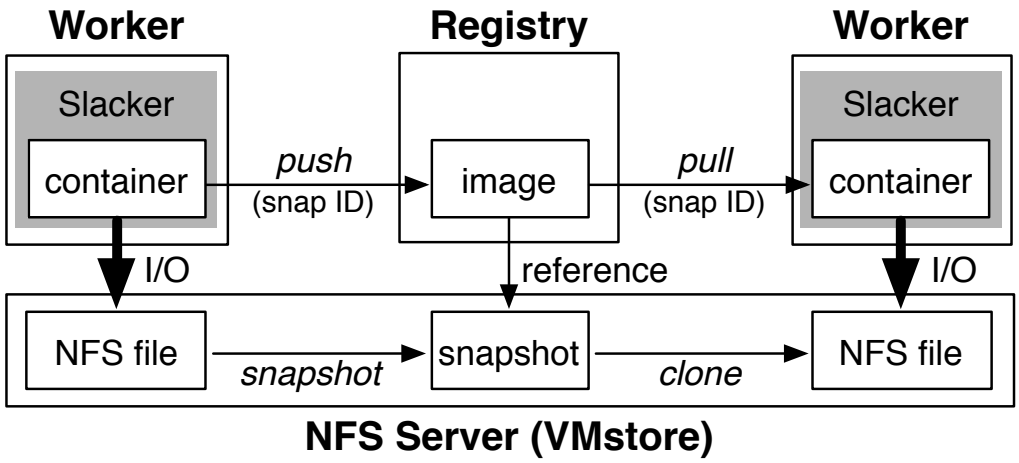

Figure 5.1: **Slacker Architecture.** *Most of our work was in the gray boxes, the Slacker storage plugin. Workers and registries represent containers and images as files and snapshots respectively on a shared Tintri VMstore server.*

will be unlikely unless container placement is highly restricted. Also, other goals (*e.g.*, load balancing and fault isolation) may make colocation uncommon. However, repeated executions are likely common for containerized utility programs (*e.g.*, python or gcc) and for applications running in small clusters. Our results suggest these latter scenarios would benefit from cache sharing.

5.2 Architectural Overview

In this section, we give an overview of Slacker, a new Docker storage driver based on our analysis.

Figure 5.1 illustrates the architecture of a Docker cluster running Slacker. The design is based on centralized NFS storage, shared between all Docker daemons and registries. Most of the data in a container is not needed to execute the container, so Docker workers only fetch data lazily from shared storage as needed. For NFS storage, we use a Tintri VMstore server [106].

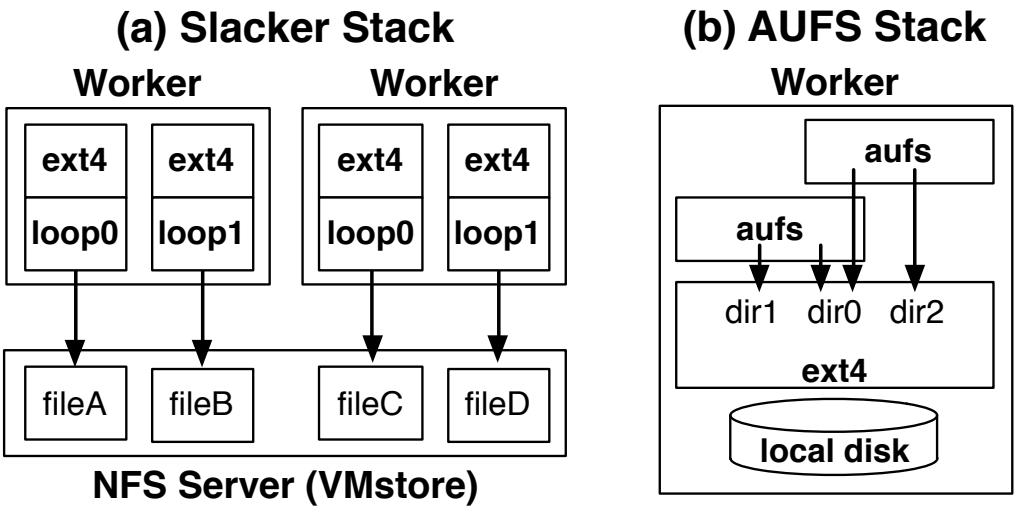

Figure 5.2: **Driver Stacks.** *Slacker uses one ext4 file system per container. AUFS containers share one ext4 instance.*

Docker images are represented by VMstore's read-only snapshots. Registries are no longer used as hosts for layer data, and are instead used only as name servers that associate image metadata with corresponding snapshots. Pushes and pulls no longer involve large network transfers; instead, these operations simply share snapshot IDs. Slacker uses VMstore `snapshot` to convert a container into a shareable image and `clone` to provision container storage based on a snapshot ID pulled from the registry. Internally, VMstore uses block-level COW to implement `snapshot` and `clone` efficiently.

5.3 Storage Layers

Our analysis revealed that only 6.4% of the data transferred by a pull is actually needed before a container can begin useful work (§4.3). In order to avoid wasting I/O on unused data, Slacker stores all container data on an NFS server (a Tintri VMstore) shared by all workers; workers lazily fetch

only the data that is needed. Figure 5.2a illustrates the design: storage for each container is represented as a single NFS file. Linux loopbacks (§5.6) are used to treat each NFS file as a virtual block device, which can be mounted and unmounted as a root file system for a running container. Slacker formats each NFS file as an ext4 file system.

Figure 5.2b compares the Slacker stack with the AUFS stack. Although both use ext4 (or some other local file system) as a key layer, there are three important differences. First, ext4 is backed by a network disk in Slacker, but by a local disk with AUFS. Thus, Slacker can lazily fetch data over the network, while AUFS must copy all data to the local disk before container startup.

Second, AUFS does COW above ext4 at the file level and is thus susceptible to the performance problems faced by layered file systems (§4.5). In contrast, Slacker layers are effectively flattened at the file level. However, Slacker still benefits from COW by utilizing block-level COW implemented within VMstore (§5.4). Furthermore, VMstore deduplicates identical blocks internally, providing further space savings between containers running on different Docker workers.

Third, AUFS uses different directories of a single ext4 instance as storage for containers, whereas Slacker backs each container by a different ext4 instance. This difference presents an interesting tradeoff because each ext4 instance has its own journal. With AUFS, all containers will share the same journal, providing greater efficiency. However, journal sharing is known to cause priority inversion that undermines QoS guarantees [117], an important feature of multi-tenant platforms such as Docker. Internal fragmentation [8, Ch. 17] is another potential problem when NFS storage is divided into many small, non-full ext4 instances. Fortunately, VMstore files are sparse, so Slacker does not suffer from this issue.

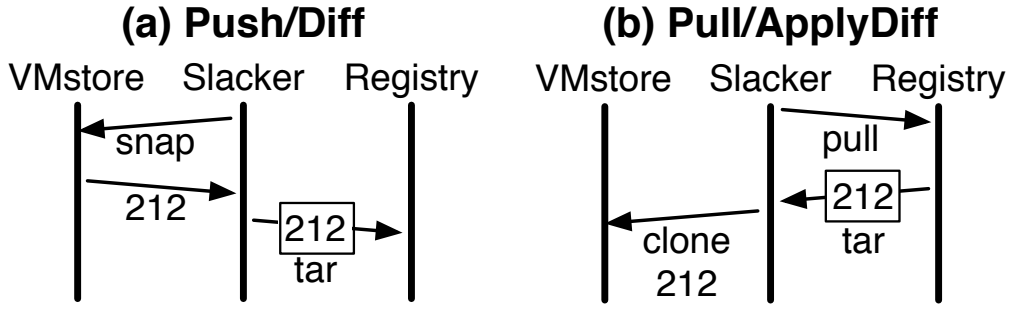

Figure 5.3: **Push/Pull Timelines.** *Slacker implements* `Diff` *and* `ApplyDiff` *with* `snapshot` *and* `clone` *operations.*

5.4 VMstore Integration

Earlier, we found that Docker pushes and pulls are quite slow compared to runs (§4.4). Runs are fast because storage for a new container is initialized from an image using the COW functionality provided by AUFS. In contrast, push and pull are slow with traditional drivers because they require copying large layers between different machines, so AUFS's COW functionality is not usable. Unlike other Docker drivers, Slacker is built on shared storage, so it is conceptually possible to use COW sharing between daemons and registries.

Fortunately, VMstore extends its basic NFS interface with an auxiliary REST-based API that, among other things, includes two related COW functions, `snapshot` and `clone`. The `snapshot` call creates a read-only snapshot of an NFS file, and `clone` creates an NFS file from a snapshot. Snapshots do not appear in the NFS namespace, but do have unique IDs. File-level snapshot and clone are powerful primitives that have been used to build more efficient journaling, deduplication, and other common storage operations [115]. In Slacker, we use `snapshot` and `clone` to implement `Diff` and `ApplyDiff` respectively. These driver functions are respectively called by Docker push and pull operations (§4.1.2).

Figure 5.3a shows how a daemon running Slacker interacts with a VMstore and Docker registry upon push. Slacker asks VMstore to create a snapshot of the NFS file that represents the layer. VMstore takes the snapshot, and returns a snapshot ID (about 50 bytes), in this case "212". Slacker embeds the ID in a compressed tar file and sends it to the registry. Slacker embeds the ID in a tar for backwards compatibility: an unmodified registry expects to receive a tar file. A pull, shown in Figure 5.3b, is essentially the inverse. Slacker receives a snapshot ID from the registry, from which it can clone NFS files for container storage. Slacker's implementation is fast because (a) layer data is never compressed or uncompressed, and (b) layer data never leaves the VMstore during push and pull, so only metadata is sent over the network.

The names "`Diff`" and "`ApplyDiff`" are slight misnomers given Slacker's implementation. In particular, `Diff(A, B)` is supposed to return a delta from which another daemon, which already has A, could reconstruct B. With Slacker, layers are effectively flattened at the file namespace level. Thus, instead of returning a delta, `Diff(A, B)` returns a reference from which another worker could obtain a clone of B, with or without A.

Slacker is partially compatible with other daemons running non-Slacker drivers. When Slacker pulls a tar, it peeks at the first few bytes of the streamed tar before processing it. If the tar contains layer files (instead of an embedded snapshot), Slacker falls back to simply decompressing instead cloning. Thus, Slacker can pull images that were pushed by other drivers, albeit slowly. Other drivers, however, will not be able to pull Slacker images, because they will not know how to process the snapshot ID embedded in the tar file.

5.5 Optimizing Snapshot and Clone

Images often consist of many layers, with over half the HelloBench data being at a depth of at least nine (§4.5). Block-level COW has inherent performance advantages over file-level COW for such data, as traversing block-mapping indices (which may be flattened) is simpler than iterating over the directories of an underlying file system.

However, deeply-layered images still pose a challenge for Slacker. As discussed (§5.4), Slacker layers are flattened, so mounting any one layer will provide a complete view of a file system that could be used by a container. Unfortunately, the Docker framework has no notion of flattened layers. When Docker pulls an image, it fetches all the layers, passing each to the driver with `ApplyDiff`. For Slacker, the topmost layer alone is sufficient. For 28-layer images (*e.g.*, jetty), the extra clones are costly.

One of our goals was to work within the existing Docker framework, so instead of modifying the framework to eliminate the unnecessary driver calls, we optimize them with *lazy cloning*. We found that the primary cost of a pull is not the network transfer of the snapshot tar files, but the VMstore clone. Although clones take a fraction of a second, performing 28 of them negatively impacts latency. Thus, instead of representing every layer as an NFS file, Slacker (when possible) represents them with a piece of local metadata that records a snapshot ID. `ApplyDiff` simply sets this metadata instead of immediately cloning. If at some point Docker calls `Get` on that layer, Slacker will at that point perform a real `clone` before the mount.

We also use the snapshot-ID metadata for *snapshot caching*. In particular, Slacker implements `Create`, which makes a logical copy of a layer (§4.1.2) with a snapshot immediately followed by a clone (§5.4). If many containers are created from the same image, `Create` will be called many times on the same layer. Instead of doing a snapshot for each `Create`, Slacker only does it the first time, reusing the snapshot ID subsequent

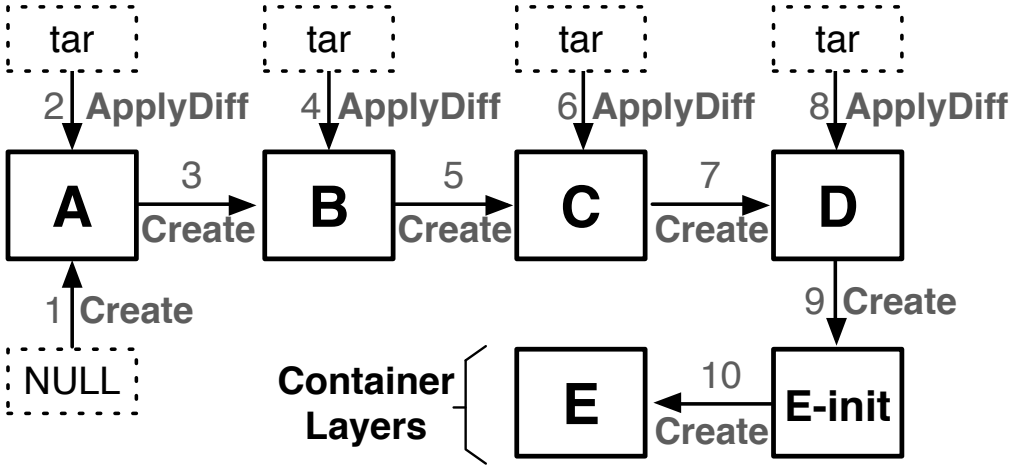

Figure 5.4: **Cold Run Example.** *The driver calls that are made when a four-layer image is pulled and run are shown. Each arrow represents a call (`Create` or `ApplyDiff`), and the nodes to which an arrow connects indicate arguments to the call. Thick-bordered boxes represent layers. Integers indicate the order in which functions are called.* ***Note:*** *this figure is identical to Figure 4.2; it is reproduced here for easy reference.*

times. The snapshot cache for a layer is invalidated if the layer is mounted (once mounted, the layer could change, making the snapshot outdated).

The combination of snapshot caching and lazy cloning can make `Create` very efficient. In particular, copying from a layer A to layer B may only involve copying from A's snapshot cache entry to B's snapshot cache entry, with no special calls to VMstore. Figure 5.4 shows the 10 `Create` and `ApplyDiff` calls that occur for the pull and run of a simple four-layer image. Without lazy caching and snapshot caching, Slacker would need to perform 6 snapshots (one for each `Create`) and 10 clones (one for each `Create` or `ApplyDiff`). With our optimizations, Slacker only needs to do one snapshot and two clones. In step 9, `Create` does a lazy clone, but Docker calls Get on the E-init layer, so a real clone must be performed. For step 10, `Create` must do both a snapshot and clone to produce and mount layer E as the root for a new container.

5.6 Linux Kernel Modifications

Our analysis showed that multiple containers started from the same image tend to read the same data, suggesting cache sharing could be useful (§4.6). One advantage of the AUFS driver is that COW is done above an underlying file system. This means that different containers may warm and utilize the same cache state in that underlying file system. Slacker does COW within VMstore, beneath the level of the local file system. This means that two NFS files may be clones (with a few modifications) of the same snapshot, but cache state will not be shared, because the NFSv3 protocol is not built around the concept of COW sharing. Cache deduplication could help save cache space, but this would not prevent the initial I/O. It would not be possible for deduplication to realize two blocks are identical until both are transferred over the network from the VMstore. In this section, we describe our technique to achieve sharing in the Linux page cache at the level of NFS files.

In order to achieve client-side cache sharing between NFS files, we modify the layer immediately above the NFS client (*i.e.*, the loopback module) to add awareness of VMstore snapshots and clones. In particular, we use bitmaps to track differences between similar NFS files. All writes to NFS files are via the loopback module, so the loopback module can automatically update the bitmaps to record new changes. Snapshots and clones are initiated by the Slacker driver, so we extend the loopback API so that Slacker can notify the module of COW relationships between files.

Figure 5.5 illustrates the technique with a simple example: two containers, B and C, are started from the same image, A. When starting the containers, Docker first creates two init layers (B-init and C-init) from the base (A). Docker creates a few small init files in these layers. Note that the "m" is modified to an "x" and "y" in the init layers, and that the zeroth bits are flipped to "1" to mark the change. Docker the creates the

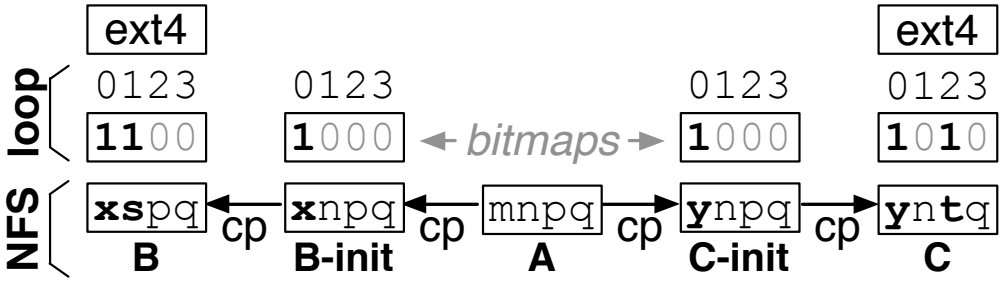

Figure 5.5: **Loopback Bitmaps.** *Containers B and C are started from the same image, A. Bitmaps track differences.*

topmost container layers, B and C from B-init and C-init. Slacker uses the new loopback API to copy the B-init and C-init bitmaps to B and C respectively. As shown, the B and C bitmaps accumulate more mutations as the containers run and write data. Docker does not explicitly differentiate init layers from other layers as part of the API, but Slacker can infer layer type because Docker happens to use an "-init" suffix for the names of init layers.

Now suppose that container B reads block 3. The loopback module sees an unmodified "0" bit at position 3, indicating block 3 is the same in files B and A. Thus, the loopback module sends the read to A instead of B, thus populating A's cache state. Now suppose C reads block 3. Block 3 of C is also unmodified, so the read is again redirected to A. Now, C can benefit from the cache state of A, which B populated with its earlier read.

Of course, for blocks where B and C differ from A, it is important for correctness that reads are not redirected. Suppose B reads block 1 and then C reads from block 1. In this case, B's read will not populate the cache since B's data differs from A. Similarly, suppose B reads block 2 and then C reads from block 2. In this case, C's read will not utilize the cache since C's data differs from A.

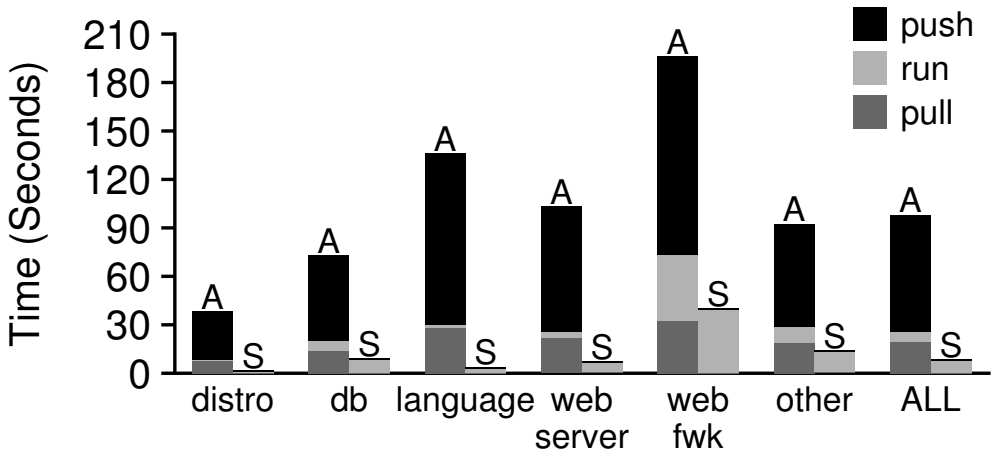

Figure 5.6: **AUFS vs. Slacker (Hello).** *Average push, run, and pull times are shown for each category. Bars are labeled with an "A" for AUFS or "S" for Slacker.*

5.7 Evaluation

We use the same hardware for evaluation as we did for our analysis. In particular, all performance measurements are taken from a virtual machine running on an PowerEdge R720 host with 2 GHz Xeon CPUs (E5-2620). The VM is provided 8 GB of RAM, 4 CPU cores, and a virtual disk backed by a Tintri T620 [105]. The server and VMstore had no other load during the experiments. For a fair comparison, we also use the same VMstore for Slacker storage that we used for the virtual disk of the VM running the AUFS experiments.

5.7.1 HelloBench Workloads

Earlier, we saw that with HelloBench, push and pull times dominate while run times are very short (Figure 4.9). We repeat that experiment with Slacker, presenting the new results alongside the AUFS results in Figure 5.6. On average, the push phase is 153× faster and the pull phase is

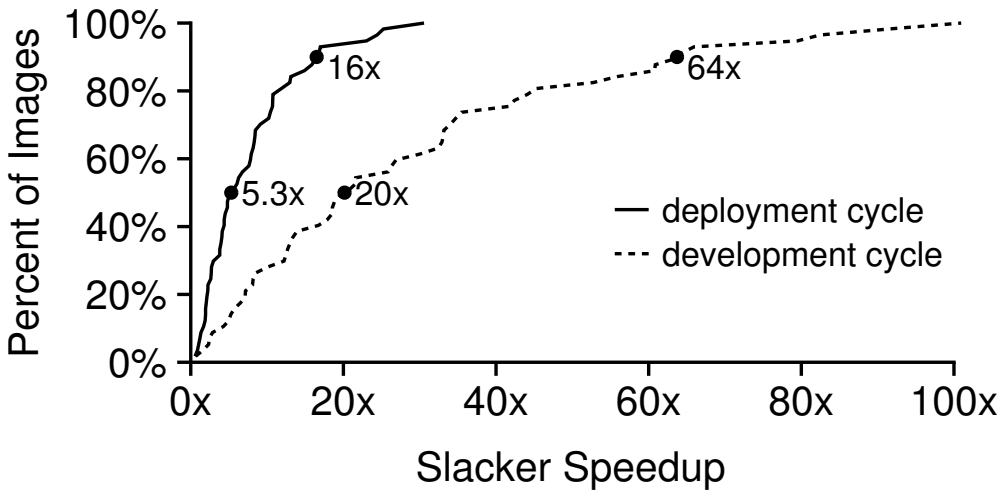

Figure 5.7: **Slacker Speedup.** *The ratio of AUFS-driver time to Slacker time is measured, and a CDF shown across HelloBench workloads. Median and 90th-percentile speedups are marked for the development cycle (push, pull, and run), and the deployment cycle (just pull and run).*

72× faster, but the run phase is 17% slower (the AUFS pull phase warms the cache for the run phase).

Different Docker operations are utilized in different scenarios. One use case is the *development cycle*: after each change to code, a developer pushes the application to a registry, pulls it to multiple worker nodes, and then runs it on the nodes. Another is the *deployment cycle*: an infrequently-modified application is hosted by a registry, but occasional load bursts or rebalancing require a pull and run on new workers. Figure 5.7 shows Slacker's speedup relative to AUFS for these two cases. For the median workload, Slacker improves startup by 5.3× and 20× for the deployment and development cycles respectively. Speedups are highly variable: nearly all workloads see at least modest improvement, but 10% of workloads improve by at least 16× and 64× for deployment and development respectively.

5.7.2 Long-Running Performance

In Figure 5.6, we saw that while pushes and pulls are much faster with Slacker, runs are slower. This is expected, as runs start before any data is transferred, and binary data is only lazily transferred as needed. We now run several long-running container experiments; our goal is to show that once AUFS is done pulling all image data and Slacker is done lazily loading hot image data, AUFS and Slacker have equivalent performance.

For our evaluation, we select two databases and two web servers. For all experiments, we execute for five minutes, measuring operations per second. Each experiment starts with a pull. We evaluate the PostgreSQL database using pgbench, which is "loosely based on TPC-B" [44]. We evaluate Redis, an in-memory database, using a custom benchmark that gets, sets, and updates keys with equal frequency. We evaluate the Apache web server, using the `wrk` [41] benchmark to repeatedly fetch a static page. Finally, we evaluate io.js, a JavaScript-based web server similar to node.js, using the `wrk` benchmark to repeatedly fetch a dynamic page.

Figure 5.8a shows the results. AUFS and Slacker usually provide roughly equivalent performance, though Slacker is somewhat faster for Apache. Although the drivers are similar with regard to long-term performance, Figure 5.8b shows Slacker containers start processing requests 3-19$\times$ sooner than AUFS.

5.7.3 Caching

We have shown that Slacker provides much faster startup times relative to AUFS (when a pull is required) and equivalent long-term performance. One scenario where Slacker is at a disadvantage is when the same short-running workload is run many times on the same machine. For AUFS, the first run will be slow (as a pull is required), but subsequent runs will be fast because the image data will be stored locally. Moreover, COW is

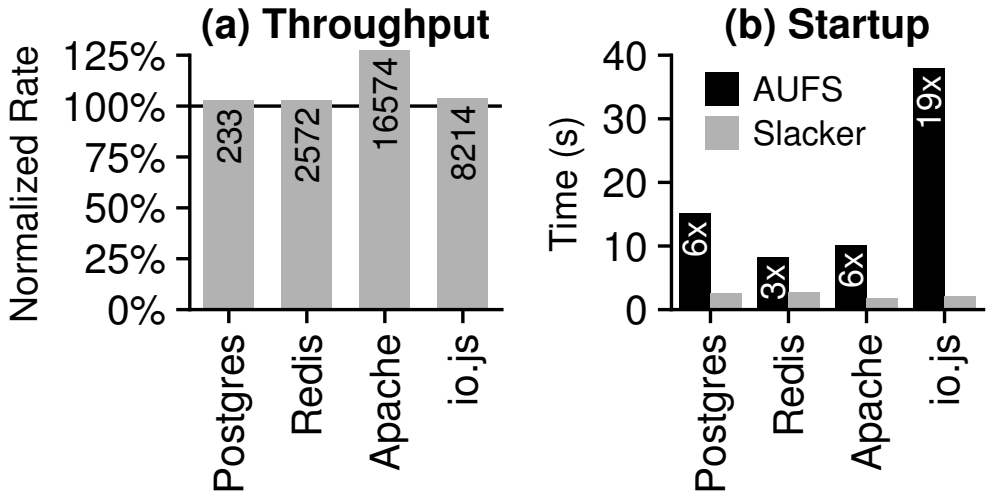

Figure 5.8: **Long-Running Workloads.** *Left: the ratio of Slacker's to AUFS's throughput is shown; startup time is included in the average. Bars are labeled with Slacker's average operations/second. Right: startup delay is shown.*

done locally, so multiple containers running from the same start image will benefit from a shared RAM cache.

Slacker, on the other hand, relies on the Tintri VMstore to do COW on the server side. This design enables rapid distribution, but one downside is that NFS clients are not naturally aware of redundancies between files without our kernel changes. We compare our modified loopback driver (§5.6) to AUFS as a means of sharing cache state. To do so, we run each HelloBench workload twice, measuring the latency of the second run (after the first has warmed the cache). We compare AUFS to Slacker, with and without kernel modifications.

Figure 5.9 shows a CDF of run times for all the workloads with the three systems (note: these numbers were collected with a VM running on a ProLiant DL360p Gen8). Although AUFS is still fastest (with median runs of 0.67 seconds), the kernel modifications significantly speed up Slacker; the median performance of Slacker alone is 1.71 seconds; with kernel modifications to the loopback module it is 0.97 seconds. Although

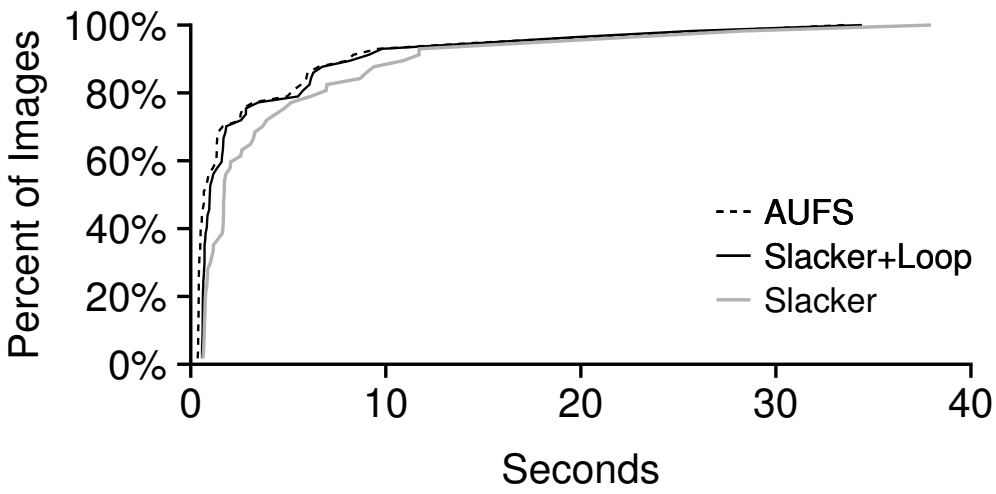

Figure 5.9: **Second Run Time (CDF).** *A distribution of run times are shown for the AUFS driver and for Slacker, both with and without use of the modified Loopback driver.*

Slacker avoids unnecessary network I/O, the AUFS driver can directly cache ext4 file data, whereas Slacker caches blocks beneath ext4, which likely introduces some overhead.

5.7.4 Scalability

Earlier (§4.4), we saw that AUFS scales poorly for pushes and pulls with regard to image size and the number of images being manipulated concurrently. We repeat our earlier experiment (Figure 4.10) with Slacker, again creating synthetic images and pushing or pulling varying numbers of these concurrently.

Figure 5.10 shows the results: image size no longer matters as it does for AUFS. Total time still correlates with the number of images being processed simultaneously, but the absolute times are much better; even with 32 images, push or pull times are at most about two seconds. It is also worth noting push times are similar to pull times for Slacker, whereas

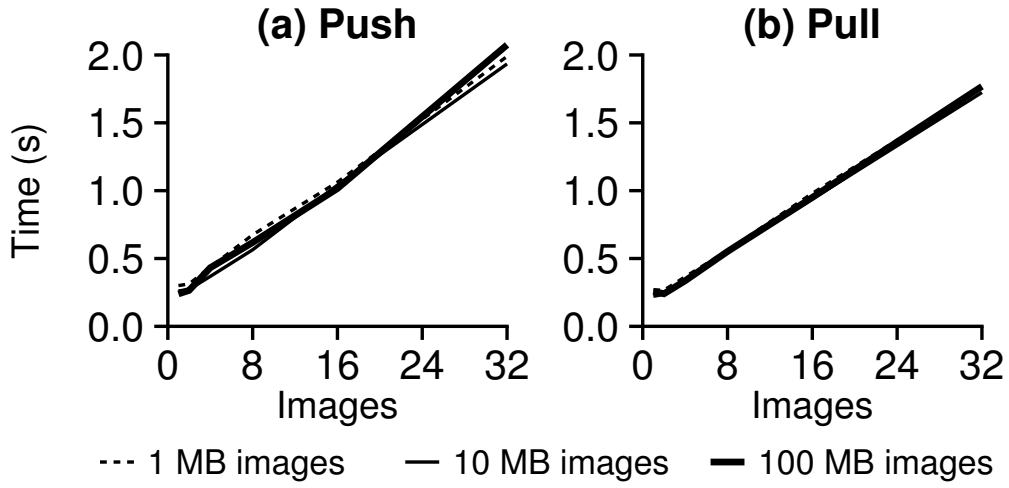

Figure 5.10: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

pushes were much more expensive for AUFS. This is because AUFS needs compression for its huge transfers, and compression is typically more costly that decompression.

5.8 Case Study: MultiMake

When starting Dropbox, Drew Houston (co-founder and CEO) found that building a widely-deployed client involved a lot of "grungy operating-systems work" to make the code compatible with the idiosyncrasies of various platforms [51]. For example, some bugs would only manifest with the Swedish version of Windows XP Service Pack 3, whereas other very similar deployments (including the Norwegian version) would be unaffected. One way to avoid some of these bugs is to broadly test software in many different environments. Several companies provide containerized integration-testing services [96, 107], including for fast testing of web ap-

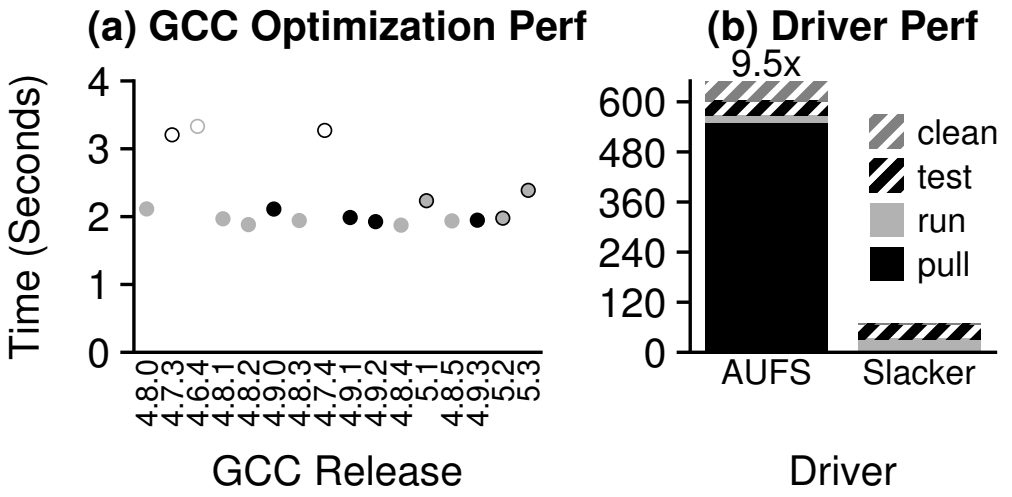

Figure 5.11: **GCC Version Testing.** *Left: run time of a C program doing vector arithmetic. Each point represents performance under a different GCC release, from 4.8.0 (Mar '13) to 5.3 (Dec '15). Releases in the same series have a common style (e.g., 4.8.* releases are solid gray). Right: performance of MultiMake is shown for both drivers. Time is broken into pulling the image, running the image (compiling), testing the binaries, and deleting the images from the local daemon.*

plications against dozens of releases of of Chrome, Firefox, Internet Explorer, and other browsers [100]. Of course, the breadth of such testing is limited by the speed at which different test environments can be provisioned.

We demonstrate the usefulness of fast container provisioning for testing with a new tool, *MultiMake*. Running MultiMake on a source directory builds 16 different versions of the target binary using the last 16 GCC releases. Each compiler is represented by a Docker image hosted by a central registry. Comparing binaries has many uses. For example, certain security checks are known to be optimized away by certain compiler releases [111]. MultiMake enables developers to evaluate the robustness of such checks across GCC versions.

Another use for MultiMake is to evaluate the performance of code snippets against different GCC versions, which employ different optimizations. As an example, we use MultiMake on a simple C program that does 20M vector arithmetic operations, as follows:

```
for (int i=0; i<256; i++) {
  a[i] = b[i] + c[i] * 3;
}
```

Figure 5.11a shows the result: most recent GCC releases optimize the vector operations well, but the 4.6.* and 4.7.* code takes about 50% longer to execute. GCC 4.8.0 produces fast code, even though it was released before some of the slower 4.6 and 4.7 releases, so some optimizations were clearly not backported. Figure 5.11b shows that collecting this data is 9.5× faster with Slacker (68 seconds) than with the AUFS driver (10.8 minutes), as most of the time is spent pulling with AUFS. Although all the GCC images have a common Debian base (which must only be pulled once), the GCC installations represent most of the data, which AUFS pulls every time. Cleanup is another operation that is more expensive for AUFS than Slacker. Deleting a layer in AUFS involves deleting thousands of small ext4 files, whereas deleting a layer in Slacker involves deleting one large NFS file.

The ability to rapidly run different versions of code could benefit other tools beyond MultiMake. For example, `git bisect` finds the commit that introduced a bug by doing a binary search over a range of commits [68]. Alongside container-based automated build systems [98], a bisect tool integrated with Slacker could very quickly search over a large number of commits.

5.9 Framework Discussion

One of our goals was to make no changes to the Docker registry or daemon, except within the pluggable storage driver. Although the storage-driver interface is quite simple, it proved sufficient for our needs. There are, however, a few changes to the Docker framework that would have enabled a more elegant Slacker implementation.

First, it would be useful for compatibility between drivers if the registry could represent different layer formats (§5.4). Currently, if a non-Slacker layer pulls a layer pushed by Slacker, it will fail in an unfriendly way. Format tracking could provide a friendly error message, or, ideally, enable hooks for automatic format conversion.

Second, it would be useful to add the notion of flattened layers. In particular, if a driver could inform the framework that a layer is flat, Docker would not need to fetch ancestor layers upon a pull. This would eliminate our need for lazy cloning and snapshot caching (§5.5).

Third, it would be convenient if the framework explicitly identified init layers so Slacker would not need to rely on layer names as a hint (§5.6).

5.10 Summary

In this chapter, we built Slacker, a new Docker storage driver, motivated by our findings in Chapter 4. Our analysis of Docker showed that startup time is dominated by pulling image data, but over 90% of the pulled data is not actually needed for startup. These findings suggest that Docker should be lazy: if most data is not needed to start a container, why should Docker wait to start a container until all that data is copied to a Docker worked from the registry?

Slacker is lazy in two ways. First, Slacker lazily allocates a file system for a new container by using the copy-on-write capabilities of a shared VMstore storage server that is used to back all containers and images. In

this regard, Slacker is not totally unlike other storage drivers. The AUFS storage driver also uses copy-on-write for allocation. However, Slacker has an advantage over AUFS in that Slacker does COW at block granularity rather than file granularity. File granularity is more costly because when data is copied upon modification, whole files must be copied. Furthermore, following COW references is expensive for layered file systems because resolution requires file-system path traversals. Slacker is more similar to the btrfs storage driver. The btrfs storage driver utilizes the block COW capabilities of the btrfs file system; the Slacker storage driver similarly utilizes the block COW capabilities of the VMstore file system.

Second, Slacker is lazy with regard to copying image data to a Docker worker in order to run a container. All prior Docker storage drivers copy all the data over the network and to local disk before a container can be started. In contrast, Slacker leverages the lazy-fetch capabilities of a local file system (*i.e.*, ext4) to fetch data from the backing server as needed.

Being lazy changes the latencies of different Docker operations. After a pull, executing "run" for the first time takes 17% longer with Slacker than with vanilla Docker because a Slacker pull fetches minimal metadata for the container whereas a Docker pull prefetches all the data the container will need to run (as well as all the data it will not). However, Slacker is able to reach a ready state much sooner because it is able to finish the pull phase and start the run phase 72x faster that vanilla Docker. The total result is that the time from the start of a pull until a container is ready to do useful work is 5x shorter with Slacker.

6 Related Work

In this chapter, we discuss other work related to our analysis and implementation efforts. We start by discussing other workload measurement studies (§6.1); the majority of our work falls into this category. Next, we consider prior techniques (some of which we simulated) for integrating the layers of a storage system (§6.2). Some of the techniques for deploying containers with Slacker were inspired by work on virtual machines. We describe research done with virtual machines and consider other techniques that could potentially be applied to containers in the future (§6.3). Finally, we describe various approaches to cache sharing between applications (§6.4).

6.1 Workload Measurement

The study of I/O workloads has a rich history in the systems community. Ousterhout *et al.* [78] and Baker *et al.* [10] collected system-call traces of file-system users. These much older studies have many findings in common with our Apple desktop and Facebook Messages studies; for example, most files are small and short lived. Other patterns (*e.g.*, frequent `fsync` calls by Apple desktop applications) are new trends.

Vogels [109] is another somewhat later study that compares results with the Ousterhout *et al.* [78] and Baker *et al.* findings. Vogels found that

files were larger than those in the earlier studies, but lifetimes were still short, and durability controls were still rarely used.

While most file-system studies deal with aggregate workloads, our examination of application-specific behaviors for the Apple study has precedent in a number of hardware studies. In particular, Flautner *et al.*'s [36] and Blake *et al.*'s [13] studies of parallelism in desktop applications bear strong similarities to ours in the variety of applications they examine. In general, they use a broader set of applications, a difference that derives from the subjects studied. In particular, we select applications likely to produce interesting I/O behavior; many of the programs they use, like the video game Quake, are more likely to exercise threading than the file system. Finally it is worth noting that Blake *et al.* analyze Windows software using event tracing, which may prove a useful tool to conduct a similar application file-system study to ours in Windows.

Kim *et al.* [58] perform a study of ten smartphone applications that has a number of similarities to our own work. They found that programmers rely heavily on user-space layers, particularly SQLite, for performing I/O. These patterns cause excessive random writes and `fsync` calls. Kim *et al.* also found durability requirements were being placed on unimportant data, such as a web cache.

A number of studies share similarities with our analysis of Facebook Messages. In our work, we compare the I/O patterns of Facebook Messages to prior GFS and HDFS workloads. Chen *et al.* [21] provides broad characterizations of a wide variety of MapReduce workloads, making some of the comparisons possible. The MapReduce study is *broad*, analyzing traces of coarse-grained events (*e.g.*, file opens) from over 5000 machines across seven clusters. By contrast, our study is *deep*, analyzing traces of fine-grained events (*e.g.*, reads to a byte) for just nine machines.

Our methodology of trace-driven analysis and simulation of Facebook Messages is inspired by Kaushik *et al.* [56], a study of Hadoop traces from

Yahoo! Both the Yahoo! study and our work involved collecting traces, doing analysis to discover potential improvements, and running simulations to evaluate those improvements.

A recent photo-caching study by Huang *et al.* [53] focuses, much like our work, on I/O patterns across multiple layers of the stack. The photo-caching study correlated I/O across levels by tracing at each layer, whereas our approach was to trace at a single layer and infer I/O at each underlying layer via simulation. There is a tradeoff between these two methodologies: tracing multiple levels avoids potential inaccuracies due to simulator oversimplifications, but the simulation approach enables greater experimentation with alternative architectures beneath the traced layer.

In addition to these studies of dynamic workloads, a variety of papers have examined the static characteristics of file systems, starting with Satyanarayanan's analysis of files at Carnegie-Mellon University [95]. One of the most recent of these examined metadata characteristics on desktops at Microsoft over a five year time span, providing insight into file-system usage characteristics in a setting similar to the home [3]. This type of analysis provides insight into long term characteristics of files that ours cannot.

6.2 Layer Integration

In our study of Facebook Messages, we simulated two techniques (§3.5) for better integration between HBase and HDFS: local compaction and combined logging. We are not the first to suggest these methods; our contribution is to quantify how useful these techniques are for the Facebook Messages workload.

Wang *et al.* [112] observed that doing compaction above the replication layer wastes network bandwidth, and implemented Salus as a solution. With Salus, new interfaces improve integration between layers: "Salus

implements active storage by blurring the boundaries between the storage layer and the compute layer." In traditional HBase, a single RegionServer is responsible for a given key range, but data is replicated beneath the RegionServer by HDFS. With Salus, the RegionServers are also replicated, and the three RegionServers coordinate to perform the same operations, each on their own single replica of a file. Our simulation findings resemble the Salus evaluation. Doing local compaction decreases network I/O while increasing disk I/O: "Salus often outperforms HBase, especially when disk bandwidth is plentiful compared to network bandwidth."

Local compaction could also be implemented based on more generic storage abstractions. For example, Spark provides an RDD (resilient distributed dataset) abstraction [118]. Externally, an RDD appears much like a regular file or dataset, but internally the system tracks the origins of an RDD, and can intelligently decide if, when, and how to materialize a replica of an RDD. For example, suppose A and B are RDDs, and $B = f(A)$. The key to the RDD abstraction is that the storage system is aware of the relationship f. Depending on partitioning, it would be possible to lazily compute the replicas of B from the replicas from A. Furthermore, if three replicas of B are to be materialized, the system has options. It could compute B from a single replica of A, then write that data over the network to three machines; alternatively, it could use additional compute to execute f on each replica of A to produce each replica of B, without causing any network I/O. One could implement local compaction, as we simulated it, by using RDD abstractions in HBase. Compaction is fundamentally a deterministic merge sort that could be executed on each replica independently. Compaction could be implemented with RDD operations such as join and sort (or similar).

We also simulated combined logging, where logs from different RegionServers arriving at the same DataNode are merged into a single stream to improve sequentiality. Combined logging and the use of dedicated

disks for transactions is an old technique that has been applied to traditional databases [31, 81].

6.3 Deployment

In Slacker, we optimized container deployment for fast startup and handling flash crowds. While this type of work is relatively new for containers, much work has been done for virtual machines, and some of our solutions are based on that work. The deployment problems for different platforms are similar, as the ext4-formatted NFS files used by Slacker resemble virtual-disk images. The strategy of lazy propagation was inspired by virtual machine optimizations. Several of the other strategies described in this section could similarly be applied to container deployment, perhaps as future work.

Hibler *et al.* [49] built Frisbee, a system for deploying virtual disks. Frisbee completes a transfer before a virtual machine starts running, but it avoids transferring unnecessary data via file-system awareness. In particular, most blocks in a disk image may be unallocated by a file system. By inspecting the blocks containing file-system metadata, Frisbee identifies these unallocated blocks and does not transfer them.

Wartel *et al.* [113] compare multiple methods of distributing virtual-machine images from a central repository (much like a Docker registry). In particular, the work explores techniques for parallel deployment of many identical virtual machines at the same time. Some of the techniques evaluated, such as binary tree distribution, could be applied to the mass deployment of containers.

Nicolae *et al.* [75] studied image deployment and found *"prepropagation is an expensive step, especially since only a small part of the initial VM is actually accessed."* These findings resemble our own: only a small part of Docker image data is actually accessed during container startup. They

further built a distributed file system for hosting virtual machine images that supports lazy propagation of VM data.

Lagar-Cavilla *et al.* [59] built a "VM fork" function that rapidly creates many clones of a running VM. Data needed by one clone is multicast to all the clones as a means of prefetch. We believe Slacker would likely benefit from similar prefetching when many containers are deployed in parallel. However, this technique would be less useful when the containers are started at different times over a longer period.

Zhe *et al.* [119] built Twinkle, a cloud-based platform for web applications that is designed to handle *"flash crowd events."* Unfortunately, virtual-machines tend to be heavyweight, as Zhe *et al.* noted: *"virtual device creation can take a few seconds."* However, Twinkle implements many optimizations for virtual machines that have not to our knowledge been used with containers. For example, Twinkle takes memory snapshots of initialized execution environments and predicts future demand for pages so it can rapidly bring an application to a ready state. Our view is that this work is heavily optimizing an inherently heavyweight platform, virtual machines. We believe many of these techniques could be reapplied to containers, an inherently lightweight platform, for extremely low-latency startup.

6.4 Cache Sharing

A number of techniques bear similarity to our strategy for sharing cache state and reducing redundant I/O. KSM (Kernel Same-page Merging) scans and deduplicates memory [7]. While this approach saves cache space, it does not prevent initial I/O. If deduplication of NFS file data in a cache is done on the client side, the problem remains that two identical blocks must both be retrieved over the network before the client can realize they are duplicates.

Xingbo *et al.* [116] build TotalCOW, a modified btrfs file system with many similarities to our work. Containers are often run over a file system without copy-on-write functionality (*e.g.*, ext4). Thus, Docker uses union file systems on top of the underlying file system in order to provide copy-on-write and avoid excessive copying. However, when the underlying storage system provides copy-on-write, a custom storage driver can utilize this functionality in Docker. For example, Slacker exposes the COW capabilities of VMstore; similarly, a btrfs driver exposes the COW capabilities of the btrfs file system. Xingbo *et al.* encountered a caching problem with btrfs similar to what we encountered: although two files might be duplicated on disk, higher layers of the storage stack do not take advantage of that fact, so redundant I/O is done, and multiple pages in the page cache are used for the same data.

Xingbo *et al.* solve this problem by building TotalCOW, a system that modifies btrfs to index cache pages by disk location, thus servicing some block reads issued by btrfs with the page cache. We encounter a very similar issue, but the same solution is not possible because the layers (*i.e.*, NFS client and NFS server) are across different machines, whereas the entire stack is on the same machine with TotalCOW. Thus, we need to rely on a smarter client that does dirty-block tracking. Also, Slacker solves the slow-pull problem, whereas TotalCOW only improves performance when container data is already local.

The migration problem has some commonalities with the deployment problem: data is moved from host to host rather than from image server to host. Sapuntzakis *et al.* [94] try to minimize the data that must be copied during VM migration much like we try to minimize the data that must be copied during container deployment. Slacker uses dirty bitmaps to track differences between different NFS files that back containers, and thus avoid fetching data for one file when identical data is cached for another file. Similarly, Sapuntzakis *et al.* modify a VMware GSX server to

use dirty bitmaps for VM images to identify a subset of the virtual-disk image blocks that must be transferred during migration.

7 Conclusions and Future Work

The design and implementation of file and storage systems has long been at the forefront of computer systems research. Innovations such as namespace-based locality [69], crash consistency via journaling [46] and copy-on-write [14, 89], scalable on-disk structures [102], distributed file systems [52, 92], and scalable storage for web services [27, 40] have greatly influenced how we manage and store data today.

Applications and systems are constantly being built and evolving, so ongoing measurement work will always be needed. In this dissertation, we explore trends in how applications use storage. As applications grow in complexity, developers are abandoning the monolithic approach and adopting a variety of strategies for decomposing the storage problem and reusing code. In our work, we consider three applications and systems, each of which take a different approach to modularity.

First, we studied Apple desktop applications that rely heavily on user-space libraries, and found that the use of libraries often resulted in expensive transactional demands on file system (Chapter 2). Second, we studied Facebook Messages, which uses an HBase-over-HDFS architecture, and found that reusing an unmodified distributed file-system layer results in excessive network I/O and random disk writes (Chapter 3). Third, we studied Docker startup workloads, and found that cold startup time is dominated by pulling data that is mostly not used during startup (Chap-

ter 4). Finally, we used our finding in the third study to redesign Docker storage and make startup latencies 5× faster (Chapter 5).

In this chapter, we summarize our three measurement studies and the design of Slacker (§7.1) and list some of our lessons learned (§7.2). Finally, we describe our plans for future work on OpenLambda (§7.3) and conclude (§7.4).

7.1 Summary

This dissertation is comprised of four parts: three workload analysis studies and an implementation project. The workloads we studied were Apple desktop applications, Facebook Messages, and Docker containers. We used our findings from the Docker study to guide the design of a new Docker storage driver, Slacker. We now summarize these four efforts.

7.1.1 Apple Desktop

Home-user applications are important today, and their importance will increase as more users store data not only on local devices but also in the cloud. Furthermore, home-user applications are interesting from a storage perspective because they rely heavily on libraries for managing and persisting data. In order to study this class of applications, we select six Apple desktop applications for analysis: iPhoto, iTunes, iMovie, Pages, Numbers, and Keynote. We build iBench, a set of scripts for executing 34 different user tasks in these applications. While running iBench, we collect system-call traces with DTrace.

We find that the use of libraries significantly impacts I/O patterns. In particular, libraries largely determine the transactional (*i.e.*, durability, atomicity, and isolation) demands placed on the file system. Anecdotally, it appears the care with which data is handled does not correspond with the developer's likely intent. For example, the Pages word processor

sometimes flushes a list of recently used files to disk, but does not flush the actual file saved. Most (and often all) of the writes are flushed for most of the iBench tasks. The vast majority of these flushes are issued via libraries. Similar patterns hold for renames which are used for atomicity.

7.1.2 Facebook Messages

The HDFS file system was originally built for MapReduce and similar workloads, but HDFS was later adopted as a substrate for HBase. In this study, we do a multilayer study of this stack, from HBase to disk. In particular, we explore whether HDFS is an effective substrate for HBase, or whether a lack of conceptual integrity in the composition of these layers leads to inefficient use of storage resources. In order to study this stack, we build a new HDFS tracing tool which we use to collect HDFS traces under Facebook Messages. We analyze these traces and use them as input to a multilayer simulator we build.

From our analysis, we find writes represent 21% of I/O to HDFS files. Further investigation reveals the vast majority of writes are HBase overheads from logging and compaction. Aside from these overheads, Facebook Messages writes are scarce, representing only 1% of the "true" HDFS I/O. Diving deeper in the stack, simulations show writes become amplified. Beneath HDFS replication (which triples writes) and OS caching (which absorbs reads), 64% of the final disk load is write I/O. This write blowup (from 1% to 64%) emphasizes the importance of optimizing writes in layered systems, even for especially read-heavy workloads like Facebook Messages.

From our simulations, we extract the following conclusions. We find that caching at the DataNodes is still (surprisingly) of great utility; even at the last layer of the storage stack, a reasonable amount of memory per node (*e.g.*, 30 GB) significantly reduces read load. We also find that a "no-write allocate" policy generally performs best, and that higher-level

hints regarding writes only provide modest gains. Further analysis shows the utility of server-side flash caches (in addition to RAM), *e.g.*, adding a 60 GB SSD can reduce latency by 3.5x.

7.1.3 Docker Containers

Microservices are becoming increasingly popular as a way to build complex applications. In our study, we focus on Docker, a tool for deploying microservices. With a microservices architecture, each microservice controls its own environment; for example, each Docker container can be deployed with its own Linux distribution and set of packages with specific versions. Of course, this design means there is much more executable data than there would be in other models. Instead of sharing a set of packages, each service has its own libraries and other files. Sacrificing sharing clearly has the potential to introduce significant overheads. Prior measurement work confirms this intuition: the service startup time in Google Borg [108] is dominated by package installation.

In order to study this problem, we build a new benchmark, HelloBench, to drive the startup of 57 different Docker containers. We analyze the data inside the HelloBench images, and trace I/O during startup to understand how the data is used. Our study shows that there is indeed a high cost to giving each container its own environment: starting a container on a new worker where the container has not been run before takes 26 seconds. 76% of that time is spent on pulling image data. Our I/O traces show that only 6.4% of that data is actually used during container startup.

7.1.4 Slacker

Our Docker measurement study showed that giving each container its own packages and environment has a high cost in terms of time and I/O

resources. Much data is transferred that is then not used. These findings suggest that microservice deployment tools such as Docker should lazily deploy services. A service should start with just the data it needs, and the other data should be copied later (if ever).

We implement this approach in Docker by building a new Docker storage driver, Slacker, backed by a shared network file system. For the network file system, we integrate with a Tintri VMstore. Slacker is lazy in two ways. First, rather than doing a deep copy of image data when a new container runs, Slacker uses VMstore to allocate space for new containers in a copy-on-write manner. Second, Slacker workers lazily fetches container data from the VMstore as needed. Slacker additionally utilizes modifications to the local Linux kernel to improve caching and optimizes certain VMstore operations to overcome deficiencies in the Docker storage interface.

The result of using these techniques is a massive improvement in the performance of common Docker operations; image pushes become 153× faster and pulls become 72× faster. Common Docker use cases involving these operations greatly benefit. For example, Slacker achieves a 5× median speedup for container deployment cycles and a 20× speedup for development cycles. We also build MultiMake, a new container-based build tool that showcases the benefits of Slacker's fast startup. MultiMake produces 16 different binaries from the same source code, using different containerized GCC releases. With Slacker, MultiMake experiences a 10× speedup.

7.2 Lessons Learned

We now describe some of the general lessons we learned while working on this dissertation.

Modularity often causes unnecessary I/O: In our studies, we considered three different ways developers decompose their applications and systems, and in all three cases, we observed the decomposition causing unnecessary I/O. In the Apple desktop applications, `fsync` and `rename` were heavily used on many small files. It is unlikely the programmer cared about these guarantees in many (or even most) cases, yet the cost of durability and atomicity was paid because of the way the library was written. In the Facebook Messages study, we showed that building a database over a generic replication layer causes additional network I/O and increases workload randomness at the disk layer. In the Docker study, we found that giving each microservice its own file-system environment means that much binary data is copied that is not needed, at least immediately.

Layers mask costs: In all our studies, we encountered cases where layering masks costs. For example, in the very first application case study we considered, Pages saves a document, and it calls a user-space function to update a 2-byte word in a document. That update function performs a read-modify-write on the chunk containing the word. The problem is that the function is called many times, so the same chunk is repeatedly overwritten. Clearly, it was not obvious to the user of the update function that so much additional I/O would be generated by the series of calls.

In the Facebook Messages study, the special handling of HBase writes makes them surprisingly expensive. At the HDFS level, the read-/write ratio is 99/1, excluding HBase compaction and logging overheads. At the disk level, the ratio is write-dominated at 36/64. Logging, compaction, replication, and caching all combine to produce this write blowup.

In the Docker study, we showed that for layered file systems, data

stored in deeper layers is slower to access. Unfortunately, we also found that Docker images tend to be deep, with at least half of file data at depth nine or greater. Even though all files appear in a single unified file system to applications and users, accesses to deep files will have unexpectedly high latencies.

Simple measurement-driven adaptations work surprisingly well: Brooks argues that conceptual integrity "dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds." Unfortunately, modern applications and systems are too complex to be built this way; instead, all the systems we studied represented the composition of work from many different minds. One might expect that the lack of conceptual integrity would make the systems studied hopelessly inefficient. Fortunately, in the Facebook and Slacker work we found relatively simple integrations and optimizations can largely compensate for the artifacts and inefficiencies of the modular approaches taken.

In the Facebook study, we simulated changes to the HBase/HDFS interfaces; in particular, we extended the HDFS interface beyond a simple file API to enable execution directly on replicas and provide special support for HBase logging. The result is a 2.7x reduction in network I/O and a 6x reduction in log-write latencies. In our Slacker work, we reused local file systems and VMstore copy-on-write capabilities to be lazy. In particular, we avoid prematurely moving and allocating the less-useful file data belonging to each microservice.

Files remain small: In our Apple and Facebook studies, we confirm a common finding in other analysis work: most accessed files are small [10]. While this could have perhaps been expected for the desktop applications, it is somewhat surprising that files are also

small for HBase. HDFS was modeled after GFS, which was designed for large files [40]. When files are large, the data-to-metadata ratio is high, justifying the one-NameNode design of GFS and HDFS. By contrast, the fact that the Messages/HBase workload is dominated by small files suggests that perhaps the single-NameNode design should be revisited.

Cold data is important: In our Facebook and Docker studies, we found a lot of data that goes unused. Two thirds of the data on the Facebook machines in our sample went untouched for the entire week-long study. This intuitively makes sense for a chat application: old conversations are not deleted, but a typical user would not have reason to revisit most message threads on a regular basis. In the Docker study, we found that over 90% of image data is not used during container startup. This also makes intuitive sense: each container has its own bundled Linux distribution, but most of the programs and libraries deployed in a generic installation would be unlikely to be used by a microservice with a very specific purpose.

These finding suggest cold data is an important part of modern workloads, with implications for both hardware and software. In particular, we believe disks will remain part of an important storage tier for the foreseeable future, and lazy techniques will be especially useful at the software level so that cold data does not translate to wasted I/O.

7.3 Future Work

The rapid pace of innovation in datacenters [11] and the software platforms within them is once again set to transform how we build, deploy, and manage online applications and services. In early settings, every application ran on its own physical machine [5, 38]. The high costs of buying

and maintaining large numbers of machines, and the fact that each was often underutilized, led to a great leap forward: virtualization [18]. Virtualization enables tremendous consolidation of services onto servers, thus greatly reducing costs and improving manageability.

However, hardware-based virtualization is not a panacea, and lighter-weight technologies have arisen to address its fundamental issues. In particular, one leading solution in this space is *containers*, a server-oriented repackaging of Unix-style processes [8, Ch. 4] with additional namespace virtualization [70, 77]. In this dissertation, we studied Docker, a platform for deploying containers (Chapter 4). While containers are arguably an improvement over virtual machines, our study and other work on Google Borg [108] shows that giving every container its own file environment can result in very slow startup times.

In our Slacker work (Chapter 5), we optimized deployment by lazily fetching data and avoiding transfers of unnecessary data. The result was a major relative improvement in deployment times (*i.e.*, a 5× speedup); however, even with our improvements, absolute times are still longer than might be desired. In particular, the median startup time with Slacker was 3 seconds, and the average was 8 seconds. While these times may be "good enough" for many applications, decreasing startup times to a small fraction of a second would open many new possibilities. Load balancers could reassign load at very fine granularity, and workers could be brought up to handle individual web requests. We believe achieving such extreme elasticity requires turning to new programming models.

One new cloud programming model, called *serverless computation*, is poised to transform the construction of modern scalable applications. Instead of thinking of applications as collections of servers, developers instead define applications with a set of functions with access to a common data store. An excellent example of this microservice-based platform is found in Amazon's Lambda [4]; we thus generically refer to this style of

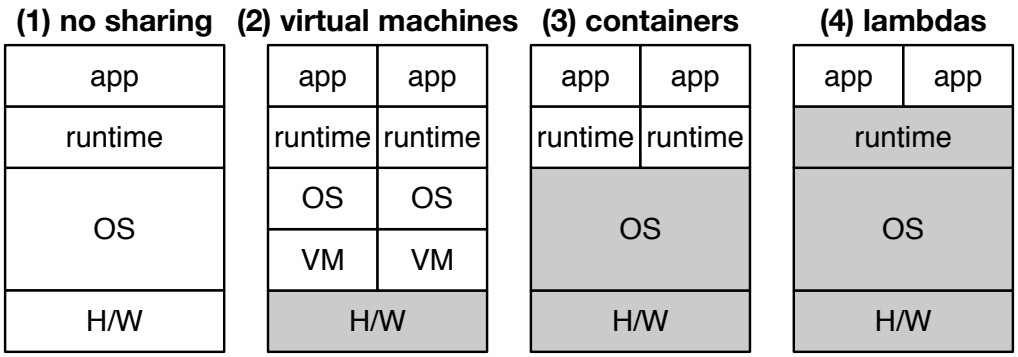

Figure 7.1: **Evolution of Sharing.** *Gray layers are shared.*

service construction as the Lambda model.

The Lambda model has many benefits as compared to more traditional, server-based approaches. Lambda handlers from different customers share common pools of servers managed by the cloud provider, so developers need not worry about server management. Handlers are typically written in languages such as JavaScript or Python; by sharing the runtime environment across functions, the code specific to a particular application will typically be small, and hence it is inexpensive to send the handler code to any worker in a cluster. Finally, applications can scale up rapidly without needing to start new servers. In this manner, the Lambda model represents the logical conclusion of the evolution of sharing between applications, from hardware to operating systems to (finally) the runtime environments themselves (Figure 7.1).

The Lambda model introduces many new challenges and opportunities for systems research. A Lambda execution engine must safely and efficiently isolate handlers. Handlers are inherently stateless, so there are many opportunities for integration between Lambda and database services. Lambda load balancers must make low-latency decisions while considering session, code, and data locality. There are further challenges in the areas of just-in-time compilation, package management, web ses-

sions, data aggregation, monetary cost, and portability. Unfortunately, most existing implementations [4, 42] (except OpenWhisk [54] and parts of Azure Functions [71]) are closed and proprietary. In order to facilitate research on Lambda architectures (including our own, and hopefully others), we are currently building OpenLambda, a base upon which researchers can evaluate new approaches to serverless computing. We describe OpenLambda and our research plan in more detail in Hendrickson *et al.* [48].

7.4 Closing Words

Applications and storage systems have become incredibly complex, and programmers cope with that complexity by modularizing and reusing software. Modern software is built on the effort of many different programmers, with a wide range of goals, personalities, ideologies, and experience. We have explored three ways developers reuse the work of other engineers: with libraries, layers, and microservices. Unfortunately, our analysis of Apple desktop applications, Facebook Messages, and Docker containers shows that modularity is not free. When subcomponents are combined to form a new storage stack, the final product lacks conceptual integrity, and unexpected and costly I/O patterns frequently emerge. As we have seen, intermediate libraries make it more difficult for a high-level programmer to communicate how data should be handled. Generic replication layers thwart distributed optimizations. Fine-grained microservices create significant provisioning costs.

While it is easy to critique the inefficient behaviors that emerge in the composed software we have studied, it is worth remembering that all these applications and systems have been quite successful. It would be naive to simply conclude that the engineers behind these systems were unaware of emergent properties or to assume that they should have built everything from scratch. Conceptual integrity is a nice property for a sys-

tem to have, but it is also expensive in terms of developer effort. Many companies have explicitly decided to prioritize developer velocity, with slogans such as *"speed wins"* [23] and *"move fast"* [65]. Patching together new products from old components is a good strategy for being first to market, and being first entails many competitive advantages, often in terms of branding and customer loyalty [85].

When should engineers prioritize development speed, and when should they prioritize conceptual integrity? Unfortunately, there may be no simple answer. In this dissertation, we have explored many of the technical aspects of this question, but a holistic conclusion about when and how to modularize must consider many business factors that we have not explored in this work. We can, however, conclude that decisions should be made with the wisdom gleaned by measurement and that measurement should be an ongoing process. As Brooks writes: *"even when an implementation is successful, it pays to revisit old decisions as the system evolves"* [61]. Measurement will always be important because both the technical and business factors impacting development decisions are constantly changing. As hardware becomes faster, breaking software into smaller components (*e.g.*, with microservices) becomes more affordable. As user bases grow, improving resource utilization via optimization and integration becomes more worthwhile.

Measurement has always been important to systems, but it has perhaps never been as important as it is now, given the rising complexity of applications. In this work, we have explored a very tiny fraction of existing storage workloads. Furthermore, new techniques will doubtless emerge for composing storage systems to solve new problems. The work of measurement is a never-ending endeavor. Our hope is that our ongoing measurement and that of others in the community will continue to illuminate the inner workings of the storage systems we rely on every day of our lives.

Bibliography

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, California, October 2006.

[2] 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.

[3] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[4] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, May 2016.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[6] Apple Computer, Inc. AppleScript Language Guide, March 2011.

[7] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.9 edition, 2014.

[9] Jens Axboe, Alan D. Brunelle, and Nathan Scott. blktrace(8) - Linux man page. `http://linux.die.net/man/8/blktrace`, 2006.

[10] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[11] Luiz Barroso and Urs Holzle. The Datacenter as the Computer. *Morgan and Claypool Synthesis Lectures on Computer Architecture*, 6, 2009.

[12] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[13] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of Thread-level Parallelism in Desktop Applications. *SIGARCH Comput. Archit. News*, 38:302–313, June 2010.

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

[15] Dhruba Borthakur. Under the Hood: Building and open-sourcing RocksDB. https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/, November 2013.

[16] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, Athens, Greece, June 2011.

[17] Frederick P. Brooks, Jr. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] 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.

[19] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 15–28, Boston, Massachusetts, June 2004.

[20] 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.

[21] Chen, Yanpei and Alspaugh, Sara and Katz, Randy. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.*, August 2012.

[22] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, 2013. USENIX.

[23] Adrian Cockcroft. Velocity and Volume (or Speed Wins). `http://flowcon.org/dl/flowcon-sanfran-2013/slides/AdrianCockcroft_VelocityAndVolumeorSpeedWins.pdf`, November 2013.

[24] Adrian Cockcroft. Migrating to Microservices. `http://gotocon.com/dl/goto-berlin-2014/slides/AdrianCockcroft_MigratingToCloudNativeWithMicroservices.pdf`, November 2014.

[25] Jonathan Corbet. Notes from a Container. `https://lwn.net/Articles/256389/`, October 2007.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, California, December 2004.

[27] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's

Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[28] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[29] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[30] Docker Hub. `https://hub.docker.com/u/library/`, 2015.

[31] IBM Product Documentation. Notes/domino best practices: Transaction logging. `http://www-01.ibm.com/support/docview.wss?uid=swg27009309`, 2013.

[32] John R. Douceur and Willian J. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pages 59–69, Atlanta, Georgia, May 1999.

[33] Daniel Ellard and Margo I. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the 17th Annual Large Installation System Administration Conference (LISA '03)*, pages 73–85, San Diego, California, October 2003.

[34] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[35] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using Latency to Evaluate Interactive System Performance. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996.

[36] Krisztián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level Parallelism and Interactive Performance of Desktop Applications. *SIGPLAN Not.*, 35:129–138, November 2000.

[37] Ford, Daniel and Labelle, François and Popovici, Florentina I. and Stokely, Murray and Truong, Van-Anh and Barroso, Luiz and Grimes, Carrie and Quinlan, Sean. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[38] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91, Saint-Malo, France, October 1997.

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

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[41] Will Glozer. Modern HTTP Benchmarking Tool. `https://github.com/wg/wrk/`, 2015.

[42] Google. Cloud Functions. `https://cloud.google.com/functions/docs/`, May 2016.

[43] Jim Gray. Tape is Dead. Disk is Tape. Flash is Disk, RAM Locality is King, 2006.

[44] PostgreSQL Global Development Group. pgbench. `http://www.postgresql.org/docs/devel/static/pgbench.html`, September 2015.

[45] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[46] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[47] Joseph L. Hellerstein. Google Cluster Data. Google Research Blog, January 2010. Posted at `http://googleresearch.blogspot.com/2010/01/google-cluster-data.html`.

[48] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, Jun 2016. USENIX Association.

[49] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, Scalable Disk Imaging with Frisbee. In *USENIX Annual Technical Conference, General Track*, pages 283–296, 2003.

[50] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[51] Drew Houston. `https://www.youtube.com/watch?t=1278&v=NZINmtuTSu0`, July 2014.

[52] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[53] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[54] IBM. OpenWhisk. `https://developer.ibm.com/openwhisk/`, May 2016.

[55] Drew Roselli Jacob, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, 2000.

[56] Rini T. Kaushik and Milind A Bhandarkar. GreenHDFS: Towards an Energy-Conserving, Storage-Efficient, Hybrid Hadoop Compute Cluster. In *The 2010 Workshop on Power Aware Computing and Systems (HotPower '10)*, Vancouver, Canada, October 2010.

[57] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, and John Wilkes. Designing for Disasters. In *FAST*, volume 4, pages 59–62, 2004.

[58] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. *Trans. Storage*, 8(4):14:1–14:25, December 2012.

[59] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[60] Butler Lampson. Computer Systems Research – Past and Present. SOSP 17 Keynote Lecture, December 1999.

[61] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[62] 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.

[63] Dave Lester. All about Apache Aurora. `https://blog.twitter.com/2015/all-about-apache-aurora`, 2015.

[64] Andrew W. Leung, Shankar Pasupathy, Garth R. 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)*, pages 213–226, Boston, Massachusetts, June 2008.

[65] Stephen Levy. Mark Zuckerberg on Facebook's Future, From Virtual Reality to Anonymity. http://www.wired.com/2014/04/zuckerberg-f8-interview/, April 2014.

[66] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[67] Macintosh Business Unit (Microsoft). It's all in the numbers... http://blogs.msdn.com/b/macmojo/archive/2006/11/03/it-s-all-in-the-numbers.aspx, November 2006.

[68] Git Manpages. git-bisect(1) Manual Page. https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html, 2015.

[69] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[70] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, Issue 239, March 2014.

[71] Microsoft. Azure Functions. https://azure.microsoft.com/en-us/services/functions/, May 2016.

[72] Kannan Muthukkaruppan. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)*, Pacific Grove, California, October 2011.

[73] namespaces(7) - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, 2013.

[74] Aravind Narayanan. Tupperware: Containerized Deployment at Facebook. http://www.slideshare.net/Docker/aravindnarayanan-facebook140613153626phpapp02-37588997, 2014.

[75] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going Back and Forth: Efficient Multideployment and Multisnapshotting on Clouds. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 147–158. ACM, 2011.

[76] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[77] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, Jul 2011.

[78] John K. Ousterhout, Herve 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)*, pages 15–24, Orcas Island, Washington, December 1985.

[79] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[80] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[81] Matt Perdeck. Speeding up database access. `http://www.codeproject.com/Articles/296523/Speeding-up-database-access-part-8-Fixing-memory-d`, 2011.

[82] John Pescatore. Nimda Worm Shows You Can't Always Patch Fast Enough. `https://www.gartner.com/doc/340962`, September 2001.

[83] Rob Pike. Another Go at Language Design. `http://www.stanford.edu/class/ee380/Abstracts/100428.html`, April 2010.

[84] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Spaces in Plan 9. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, EW 5, pages 1–5, New York, NY, USA, 1992. ACM.

[85] Michael E Porter. *Competitive Strategy: Techniques for Analyzing Industries and Competitors*. Simon and Schuster, 2008.

[86] The Linux Information Project. Linus Torvalds: A Very Brief and Completely Unauthorized Biography. `http://www.linfo.org/linus.html`, Aug 2006.

[87] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. In *Proceedings of the 4th ACM Symposium on Operating Systems Principles (SOSP '73)*, Yorktown Heights, New York, October 1973.

[88] 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)*, pages 41–54, San Diego, California, June 2000.

[89] 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.

[90] David Saff and Michael D. Ernst. An Experimental Evaluation of Continuous Testing During Development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.

[91] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[92] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[93] Eric Sandeen. Linux Filesystems LOC. http://sandeen.net/wordpress/computers/linux-filesystems-loc/, Jun 2011.

[94] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, December 2002.

[95] Mahadev Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, California, December 1981.

[96] Sunil Shah. Integration Testing with Mesos, Chronos and Docker. `http://mesosphere.com/blog/2015/03/26/integration-testing-with-mesos-chronos-docker/`, 2015.

[97] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[98] Matt Soldo. Upgraded Autobuild System on Docker Hub. `http://blog.docker.com/2015/11/upgraded-autobuild-docker-hub/`, 2015.

[99] Nicolas Spiegelberg. Allow Record Compression for HLogs. `https://issues.apache.org/jira/browse/HBASE-8155`, 2013.

[100] spoon.net. Containerized Selenium Testing. `https://blog.spoon.net/running-a-selenium-grid-using-containers/`, 2015.

[101] SQLite. SQLite: Frequently Asked Questions. `http://www.sqlite.org/faq.html`, February 2012.

[102] Adan 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.

[103] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[104] Matthew Tilmann. Apple's Market Share In The PC World Continues To Surge. maclife.com, April 2010.

[105] Tintri. Tintri VMstore(tm) T600 Series. `http://www.tintri.com/sites/default/files/field/pdf/document/t600-datasheet_0.pdf`, 2013.

[106] Tintri. Tintri Operating System. https://www.tintri.com/sites/default/files/field/pdf/whitepapers/Tintri-OS-Datasheet-160517T10072.pdf, 2016.

[107] Paul van der Ende. Fast and Easy Integration Testing with Docker and Overcast. http://blog.xebia.com/2014/10/13/fast-and-easy-integration-testing-with-docker-and-overcast/, 2014.

[108] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[109] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[110] 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.

[111] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[112] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Presented as part of the*

10th USENIX Symposium on Networked Systems Design and Implementation, Lombard, Illinois, April 2013.

[113] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 112–117. IEEE, 2010.

[114] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[115] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[116] Xingbo Wu, Wenguang Wang, and Song Jiang. TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 15:1–15:7, New York, NY, USA, 2015. ACM.

[117] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 474–489, New York, NY, USA, 2015. ACM.

[118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, California, April 2010.

[119] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *INFOCOM, 2011 Proceedings IEEE*, pages 802–810. IEEE, 2011.