REPRESENTATIVE, REPRODUCIBLE, AND PRACTICAL BENCHMARKING OF FILE AND STORAGE SYSTEMS

by

Nitin Agrawal

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2009

To my parents

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisors, Andrea and Remzi; I couldn't have asked for better. During our weekly meetings, more often than not, I went in with a laundry list of frenzied thoughts and messy graphs, and came out with a refined idea. Graciously, they made me feel as if I was the one responsible for this outcome; I wish I had been better prepared to get more out of our meetings.

Remzi showed me how someone could do serious research without taking oneself too seriously, though it will be a lasting challenge for me to come anywhere close to how effortlessly he does it. As an advisor, he genuinely cares for his student's well being. On many occasions my spirits got refreshed after getting an "are you OK?" email from him. Remzi not only provided a constant source of good advice ranging from conference talk tips to tennis racquet selection, but his ability to find "altitude" in my work when none existed transformed a series of naive ideas into a dissertation.

Most students would consider themselves lucky to have found one good advisor, I was privileged to have two. Andrea constantly raised the bar and challenged me to do better; the thesis is all the more thorough due to her persistence. I was always amazed how she could understand in a few minutes the implications of my work far beyond my own understanding. Andrea's meticulous advising was the driving force behind the completion and acceptance of much of my work.

I would like to thank my thesis committee members Mike Swift, Karu Sankaralingam and Mark Craven, for their useful suggestions and questions about my research during my defense talk. I would especially like to thank Mike for his advice and interest in my work

throughout, and for giving me an opportunity to teach his undergraduate operating systems class.

During my six years in Wisconsin, I have had a wonderful set of colleagues Leo Arulraj, Lakshmi Bairavasundaram, John Bent, Nate Burnett, Tim Denehy, Haryadi Gunawi, Todd Jones, Shweta Krishnan, Andrew Krioukov, Joe Meehan, Florentina Popovici, Vijayan Prabhakaran, Abhishek Rajimwale, Meenali Rungta, Muthian Sivathanu, Sriram Subramanian, Swami Sundararaman, and Yupu Zhang; being the last of Remzi and Andrea's pre-tenure hires, I perhaps overlapped with a larger than usual group of students to my benefit. I will especially miss our late night chats in the hallway and lunch outings to the food carts. Muthian and Ina kindled my interest in the group and provided a patient ear to a curious (and clueless) first year graduate student, I thank them for that. I also thank Leo for taking over some of the work on Compressions while I was busy writing my dissertation. My officemates John, Todd, Shweta, and Swami have all been awesome (even taking care of my plants during my absence!).

I have greatly benefited from my internships at Microsoft Research (Redmond and Silicon Valley) and IBM Almaden; my internship work at MSR Redmond in fact became the basis for my thesis itself. I thank my mentors and colleagues Bill Bolosky, John Douceur, Jay Lorch at MSR Redmond, Ted Wobber at MSR Silicon Valley, and Ying Chen at IBM Almaden. I am grateful to John and Ted in particular for providing a terrific internship experience as mentors.

Gautham and Lakshmi, my roommates for five years, made my stay in Madison a most pleasant one. Their friendship and camaraderie provided comfort during times when I felt low, and a reason to celebrate even minor accomplishments. I will miss my discussions with Gau on all things not related to work. Lakshmi had the added burden of being both a friend and a colleague. He mentored my research in more ways than he cares to remember. I thank Anoop for being part of our Madison family during the first two years. Having

Akash as roommate for the last year during a tough job market somehow made the process less stressful.

My stay in Madison wouldn't have been the same without the company of wonderful friends Ankit, Cindy, Jayashree, Neelam, Raman, Shweta and Swami to name a few. I constantly ignored Shweta and Neelam's concern about my health and eating habits, but they continued to provide food and friendship during their years here. Evening coffee breaks with them became a daily stress buster, and our escapades during a summer internship in California will always be memorable. Cindy not only provided constant company in the department, but also pleasant musical interludes. Having Ankit live right next door made his apartment my "third place"; movie and dinner nights eventually gave way to more entertaining ones playing Wii.

Mayur, and later on Priyank, played host to numerous east coast trips; knowing that they will be close to Princeton is a comforting thought. Deborshi and Siddhartha have been good friends throughout these years. The company of my cousins in the US, Arvind, Ashish, Saurabh and Shubha, provided me a feeling of home away from home.

My sister Neha has always been the kind of loving sister who is more happy for my accomplishments than I myself. Talking to her on phone has been my refuge from the monotony of work and I have come to cherish the time that we get to spend together. Finally, I would like to thank my parents for their unconditional love and support; this Ph.D. would not have been possible without them. They have always helped me see the good in everything and shaped my perspective for the better. Whenever I doubted my abilities, their encouragement put me right back on track. The pride in their eyes on seeing me graduate made it all seem worthwhile.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

REPRESENTATIVE, REPRODUCIBLE, AND PRACTICAL BENCHMARKING OF FILE AND STORAGE SYSTEMS

Nitin Agrawal

Under the supervision of Professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

At the University of Wisconsin-Madison

Benchmarks are crucial to assessing performance of file and storage systems; by providing a common measuring stick among differing systems, comparisons can be made, and new techniques deemed better or worse than existing ones. Unfortunately, file and storage systems are currently difficult to benchmark; there is little consensus regarding the workloads that matter and insufficient infrastructure to make it easy to run interesting workloads. This dissertation attempts to simplify the task of file and storage system benchmarking by focusing on three of its important principles– developing an understanding of and creating solutions for *representative*, *reproducible* and *practical* benchmarking state and benchmark workloads.

We first develop an understanding of file-system metadata that comprises much of the file-system state by performing the first large-scale longitudinal study of file system snapshots. We then develop means to recreate representative and reproducible file-system state for benchmarking by building *Impressions*, a framework to generate statistically accurate file-system images with realistic metadata and content. We develop a system *Compressions*, that makes it practical to run large, complex benchmarks on storage systems with modest capacities, while also being faster in total runtime if desired. We also develop an understanding towards creating representative, reproducible and practical synthetic benchmark workloads, and describe our first steps in creating "realistic synthetic" benchmarks by building a tool called *CodeMRI*.

Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

ABSTRACT

Benchmarks are crucial to assessing performance of file and storage systems; by providing a common measuring stick among differing systems, comparisons can be made, and new techniques deemed better or worse than existing ones. Unfortunately, file and storage systems are currently difficult to benchmark; there is little consensus regarding the workloads that matter and insufficient infrastructure to make it easy to run interesting workloads. This dissertation attempts to simplify the task of file and storage system benchmarking by focusing on all three of its important principles– developing an understanding of and creating solutions for *representative*, *reproducible* and *practical* benchmarking state and benchmark workloads.

We develop an understanding of file-system metadata by performing a large-scale longitudinal study of file system snapshots representative of corporate PC systems. For five years from 2000 to 2004, we collected annual snapshots of file-system metadata from over 60,000 Windows PC file systems in a large corporation. In our study, we use these snapshots to study temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity and consumption, and degree of file modification. We present a generative model that explains the namespace structure and the distribution of directory sizes. We find significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities.

We develop means to recreate *representative* and *reproducible* file-system state for benchmarking. The performance of file systems and related software depends on characteristics of underlying file-system image (*i.e.*, file-system metadata and file contents). Unfortunately, rather than benchmarking with realistic file-system images, most system designers and evaluators rely on *ad hoc* assumptions and (often inaccurate) rules of thumb. To remedy these problems, we develop *Impressions*, a framework to generate statistically accurate file-system images with realistic metadata and content; we present its design, implementation and evaluation. Impressions is flexible, supporting user-specified constraints on various file-system parameters using a number of statistical techniques to generate consistent images. We find that Impressions not only accurately quantifies benchmark performance, but also helps uncover application policies and potential bugs, making it a useful for system developers and users alike.

We develop a system that makes it *practical* to run large, complex benchmarks on storage systems with modest capacities. Typically, benchmarking with such benchmarks on large disks is a frequent source of frustration for file-system evaluators; the scale alone acts as a strong deterrent against using larger albeit realistic benchmarks. To address this problem, we have developed Compressions, a benchmarking system that makes it practical to run benchmarks that were otherwise infeasible on a given system, while also being faster in total runtime. Compressions creates a “compressed” version of the original file-system image on disk by omitting all file data and laying out metadata more efficiently; we present the design, implementation and evaluation of Compressions.

We develop an understanding towards creating *representative*, *reproducible* and *practical* synthetic benchmark workloads. Synthetic benchmarks are accepted and widely used as substitutes for more realistic and complex workloads in file systems research, however, they are largely based on the benchmark writer’s interpretation of the real workload, and how it exercises the system API. It is our hypothesis that if two workloads execute roughly the same set of function calls within the file system, that they will be roughly equivalent to one another; based on this hypothesis, we describe our first steps in creating “realistic synthetic” benchmarks by building a tool called CodeMRI. CodeMRI leverages file-system domain knowledge and a small amount of system profiling in order to better understand how the benchmark is stressing the system and to deconstruct its workload.

Chapter 1

Introduction

Everyone cares about data, from scientists running simulations to families storing photos and tax returns. Thus, the file and storage systems that store and retrieve our data play an essential role in our computer systems. To handle the different needs of various user communities, many different file and storage systems have been developed, from the Google File System [50], IBM GPFS [122] and NetApp Data ONTAP storage system [45] in the enterprise segment, to local file systems such as NTFS [135] and Linux ext3 [150].

Modern file systems are provisioned with features going well beyond their primary objective of just reading and writing to storage media. For example, scalability has been a focus for file system designers to support both large counts of files and directories, and large file sizes [111, 141]. Many have investigated ways to build useful search functionality within and outside the file system, making more sophisticated use of dynamic file system usage information [52, 53, 126, 137]. Providing content management for sharing digital media across administrative domains has been explored in the context of file systems [48, 152]. Finally, reliability schemes to handle media failures both in desktop [106, 154] and enterprise systems [20] have also been developed. With this growth in file system functionality and correspondingly in complexity, the techniques to effectively evaluate the performance of file and storage systems have become increasingly more relevant.

Unfortunately, file and storage systems are currently difficult to benchmark [145]. There is little consensus regarding the workloads that matter and insufficient infrastructure to make it easy to run interesting workloads. Years of research on the design and implementation of file systems, and on applications using these file systems, has led to an abundance of innovations; however, approaches for benchmarking still lag behind. Benchmark performance measurements are often uncorrelated with representative real-world settings and the results are hard to compare across systems due to lack of standardization and reproducibility. This general state of disarray in file-system benchmarking complicates matters for system designers, evaluators, and users alike.

The objective of this dissertation is to simplify the task of benchmarking by developing tools and techniques that provide a thorough and easy to use experience for file and storage system benchmarking. To do so, we focus on three important principles that systems benchmarking should adhere to, and is currently lacking in being:

- **Representative:** of target usage scenario and real-world conditions
- **Reproducible:** to provide a common yardstick and enable fair comparison
- **Practical:** to encourage community-wide adoption and standardization

In order to meet the expectations defined by these principles, we have set ourselves the following goals: first, to examine the characteristics and develop an understanding of file system metadata that forms much of the basis for representative file system state; second, to provide means to adopt and accurately reproduce the empirical information about file system metadata for experimental usage by creating representative, reproducible file-system benchmarking state; third, to make it practical to run complex, real-world benchmarks on realistic storage infrastructure; and finally, also develop an understanding of workloads used for file system benchmarking and provide means to create representative, reproducible and practical synthetic benchmark workloads.

We address the goals of this dissertation as follows. First, we analyze the transverse and longitudinal properties of file system metadata by conducting a large-scale study of file-system contents [8, 9]. Second, we develop a statistical framework that allows one to incorporate realistic characteristics of file system metadata and file data, and reproduce them for benchmarking [6]. Third, we develop a practical benchmarking system that allows one to run large, complex workloads with relatively modest storage infrastructure [5]. Finally, we also outline a methodology to create synthetic benchmarks that are functionally equivalent to and representative of real workloads [3, 4].

By freely contributing our data and software to the community, we encourage file-system developers to use them as standardized resources for benchmarking and experimentation. The ensuing sections explain in greater detail each of these contributions of the dissertation.

1.1 Representative and Reproducible File-System Benchmarking State

One of the primary objectives of benchmarking is to be able to evaluate the performance of a system under operating conditions that closely reflect the real-world scenario in which the system-under-test is going to be deployed. Another important objective is to allow comparison among competing systems; for a fair comparison, the performance of two or more systems should be measured *under the same operating conditions*. By providing a common measuring stick among differing systems, benchmarks can quantifiably distinguish performance, and allow new techniques to be deemed better or worse than existing ones.

Thus, any benchmarking experiment must be preceded by an initialization phase to recreate the necessary state representative of the target usage scenario. The state should itself be reproducible, so as to enable different developers to ensure they are all comparing performance of a system under the same conditions.

For file and storage systems, generating representative and reproducible state for benchmarking requires first an understanding of the file system state which primarily consists of the file-system metadata, and second, means to create an initial file-system image (or benchmarking state) that is both representative of the metadata properties and reproducible for experimentation. Throughout this document, we refer to a *benchmark* as the combination of a *benchmark workload* and the *benchmark state*.

1.1.1 Characteristics of File-System Metadata

File systems store user data and its associated *metadata* or bookkeeping information on a storage device such as a hard disk. Metadata is stored in an internal representation to allow operations such as creation, look up, insertion and deletion of user data. The *on-disk* image of the file system contains this information in a persistent data structure; for example, NTFS [135] and ReiserFS [111] use balanced B+ trees, Apple's HFS [15] and Reiser4 [79] use B^* trees, while Ext3 [150] uses statically reserved block groups to store metadata, similar to cylinder groups in the Berkeley FFS [83]. Operations performed on metadata heavily influence the overall performance of a file system; metadata intensive benchmarks are often considered a stress test for any file system.

Detailed knowledge of file-system metadata is thus essential for designing file and storage systems, and in our case, for creating representative benchmark state. Real world information about the structure, organization and contents of files and directories stored in file systems is crucial both to make informed design decisions and to accurately evaluate these systems. However useful, real world data is hard to collect and analyze, forcing storage system designers to rely on anecdotal information and often outdated rules of thumb.

In the past, there have been few empirical studies of file-system metadata [67, 94, 120, 128], and even fewer that involved any sizeable metadata collection [41]. In this dissertation, we present the first large-scale longitudinal study of file-system metadata. To perform this study, every year from 2000 to 2004, we collected snapshots of metadata from over ten thousand file systems on Windows desktop computers at Microsoft Corporation. Our resulting data sets contain metadata from 63,398 distinct file systems, 6457 of which provided snapshots in multiple years. These systems contain 4 billion files totaling 700 TB of file data, allowing us to understand various characteristics of file-system metadata and how it evolves over a multi-year time period.

In particular, we studied temporal changes in the size, age, and type frequency of files, the size of directories, the structure of the file-system namespace, and various characteristics of file systems, including file and directory population, storage capacity, storage consumption, and degree of file modification. Our measurements revealed several interesting properties of file systems and offered useful lessons. In this study we find significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities.

One interesting discovery is the emergence of a second mode in the Gigabyte range in the distribution of bytes by containing file size. A few large files, mainly video, database, and blob files, are the dominant contributors to this second mode and are responsible for an increasing fraction of the total file-system usage. The increasingly large fraction of content in large files suggests that variable block sizes, as supported by ZFS [24] and NTFS [136], are becoming increasingly important.

Another interesting finding is our observation on file system fullness changes. Over the course of our five-year study, despite a vast increase in available file-system capacity (roughly 500% increase in the arithmetic mean of available capacity), aggregate file system fullness remained remarkably stable at 41% over all years, and mean fullness dropped only by 4%. Storage manufacturers can thus keep focusing effort on increasing capacity, because customers will continue to place great value on capacity for the foreseeable future.

We have made our dataset available to the community via the Storage Networking Industry Association's IOTA repository. To obtain it, visit the URL `http://iotta.snia.org/traces/tracesStaticSnapshot/`. We hope this will help others use this dataset in their own endeavors and also encourage more such studies to be conducted in the future.

1.1.2 Generating File-System Benchmarking State

Armed with the information on file-system metadata, the second requirement in order to generate file-system state for benchmarking is to develop means to create an initial file-system image that is both representative of the metadata properties and reproducible for experimentation.

In order for a benchmark execution to be comprehensible, the system under test needs to be brought to a known state before running the benchmark workload. Typically, this initialization consists of a warm-up phase wherein the benchmark workload is run on the system for some time prior to the actual measurement phase. An example of such a warm-up phase is warming the caches prior to evaluating performance of memory organization in processors [63], database systems [88] or file systems. Previous work has noted that the contents of the cache during a test have significant impact on the performance results [25, 38, 63]. Different systems may have additional requirements for the initialization phase, but across systems, state is important when doing performance evaluation.

Several factors contribute to the file-system state, important amongst them are the *in-memory* state (contents of the file-system buffer cache), the *on-disk* state (disk layout and fragmentation) and the characteristics of the *file-system image* (files and directories belonging to the namespace and file contents).

Cache warm-up takes care of the in-memory state of file systems. Another important factor is the on-disk state of the file system, or the degree of *fragmentation*; it is a measure of how the disk blocks belonging to the file system are laid out on disk. Previous work has shown that fragmentation can adversely affect performance of a file system [132]. Thus, prior to benchmarking, a file system should undergo *aging* by replaying a workload similar to that experienced by a real file system over a period of time [132].

In the case of file and storage system benchmarking, the requirement for reproducibility translates to the need for methodically recreating file-system state prior to every benchmark run. This would include warming up the caches, inducing appropriate fragmentation for on-disk layout as mentioned before, and creating a *file-system image* representative of the target environment.

Surprisingly, the characteristics of the file-system image, a key contributor to file-system state have been largely ignored in creating benchmarking state; the properties of file-system metadata and file content can have a significant impact on the performance of a system. Properties of file-system metadata includes information on how directories are organized in the file-system namespace, how files are organized into directories, and the distributions for various file attributes such as size, depth, and extension type.

Much of this information can be obtained in limited forms from various empirical studies of file-system contents. Such studies focus on measuring and modeling different aspects of file-system metadata by collecting snapshots of file-system images from real machines. Collecting and analyzing this data provides useful information on how file systems are used in real operating conditions.

However, no clear methodology exists to incorporate this knowledge in accurately and reproducibly creating file-system images, failing which, more often than not, benchmarking resorts to arbitrary approaches for creating file-system state. The lack of standardization and reproducibility of these choices makes it almost impossible to compare results, sacrificing the utility of the benchmark itself. To address this problem, we need a systematic approach to creating benchmarking state for file systems, with particular emphasis given to creating realistic and reproducible file-system images.

With the goal of simplifying the process of creating benchmark state, we develop *Impressions*, a framework to generate statistically accurate file-system images with realistic file-system metadata and file content. Impressions is flexible in accommodating a number of user-specified inputs and supports additional constraints on various file-system parameters, using a number of statistical techniques to generate consistent file-system images.

A casual user looking to create a representative file-system image without worrying about selecting parameters can simply run Impressions with its default settings; Impressions will use pre-specified distributions from file-system studies to create a representative image. A more sophisticated user has the power to individually control the knobs for a comprehensive set of file-system parameters; Impressions will carefully work out the statistical details to produce a consistent and accurate image. In both cases, Impressions ensures complete reproducibility of the image, by reporting the used distributions, parameter values, and seeds for random number generators.

In our experiences with Impressions, we found it effective and easy to use for benchmarking file systems and related software. In a case study of popular desktop search tools in Linux, Beagle and Google Desktop, Impressions allowed us to accurately quantify and compare the overhead in building the initial search index for various indexing schemes and file-system images.

We believe Impressions will prove to be a useful tool for benchmarking and we have made it publicly available; to obtain Impressions, visit the URL `http://www.cs.wisc.edu/adsl/Software/Impressions/`.

1.2 Practical Benchmarking for Large, Real Workloads

So far we have discussed two important challenges in file-system benchmarking: recreating benchmarking infrastructure representative of real-world conditions, and ensuring reproducibility of the benchmarking state to allow fair comparison. The time and effort required to ensure that the above conditions are met often discourages developers from using benchmarks that matter, settling instead for the ones that are easy to set up and use. These deficiencies in benchmarking point to a thematic problem – when it comes to actual usage, ease of use and practicality often overshadow realism and accuracy.

In practice, realistic benchmarks (and realistic configurations of such benchmarks) tend to be much larger and more complex to set up than their trivial counterparts. File system traces (*e.g.*, from HP Labs [113]) are good examples of such workloads, often being large and unwieldy. In many cases the evaluator has access to only a modest infrastructure, making it harder still to employ large, real workloads.

Two trends further exacerbate this difficulty in file and storage benchmarking. First, storage capacities have seen a tremendous increase in the past few years; Terabyte-sized disks are now easily available for desktop computers; enterprise systems are now frequently dealing with Petabyte-scale storage. Second, popular applications are taking increasingly longer to execute. Examples of such applications include file-system integrity checkers like `fsck` and desktop search indexing, each taking anywhere from several hours to a few days to run on a Terabyte-sized partition.

Benchmarking with such applications on large partitions is a frequent source of frustration for file-system evaluators; the scale alone acts as a strong deterrent against using larger albeit realistic benchmarks [146]. Given the rate at which storage capacities are increasing, running toy workloads on small disks is no longer a satisfactory alternative. One obvious solution is to continually upgrade one's storage infrastructure. However, this is an expensive, and perhaps an infeasible solution, especially to justify the costs and administrative overheads solely for benchmarking.

In order to encourage developers of file systems and related software to adopt larger, more realistic benchmarks and configurations, we need means to make them practical to run on modest storage infrastructure. To address this problem, we have developed Compressions, a "scale down" benchmarking system that allows one to run large, complex workloads using relatively small storage capacities by scaling down the storage requirements transparent to the workload. Compressions makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system.

They key idea in Compressions is to create a "compressed" version of the original file-system image for the purposes of benchmarking. In the compressed image, unneeded user data blocks are omitted and file system metadata blocks (e.g., inodes, directories and indirect blocks) are laid out more efficiently on disk. To ensure that applications and benchmark workloads remain unaware of this interposition, Compressions synthetically produces file data using a suitably modified in-kernel version of Impressions, and appropriately fetches the redirected metadata blocks. Compressions then uses an in-kernel model of the disk and storage stack to determine the runtime of the benchmark workload on the original uncompressed image. The storage model calculates the run times of all individual requests as they would have executed on the uncompressed image.

Depending on the workload and the underlying file-system image, the size of the compressed image can range anywhere from 1 to 10% of the original, a huge reduction in the required disk size for benchmarking. Compressions also reduces the time taken to run the benchmark by avoiding a significant fraction of disk I/O and disk seeks. The storage model within Compressions is fairly accurate in spite of operating in real-time, and imposes an almost negligible overhead on the workload execution.

Compressions thus allows one to run benchmark workloads that require file-system images orders of magnitude larger than the available disk and to run much faster than usual, all this while still reporting the runtime as it would have taken on the original image; we believe Compressions provides a practical approach to run large, real workloads with a modest overhead and virtually no extra expense in frequently upgrading storage infrastructure for benchmarking.

1.3 Representative, Reproducible and Practical Benchmark Workloads

The other crucial requirement for benchmarking apart from the benchmark state is the benchmark workload: no benchmarking can proceed without one. Like benchmarking state, the benchmark workload should also be representative of the target usage scenario, in this case the real-world applications running on the system. While creating a benchmark workload care must be taken to ensure that the workload is easy to reproduce so as to enable comparison across systems.

To evaluate the performance of a file and storage system, developers have a few different options, each with its own set of advantages and disadvantages.

- **Real Applications:** One option for evaluating a file or storage system is to directly measure its performance when running real I/O-intensive applications.
- **Microbenchmarks of Application Kernels:** A second option is to run application kernels that are simplified versions of the full applications themselves.
- **Trace replay:** A third option is to replay file system traces that have been previously gathered at various research and industrial sites.
- **Synthetic workloads:** A final option is to run synthetic workloads that are designed to stress file systems appropriately, even as technologies change.

On the whole, synthetic benchmark workloads are much more popular than real workloads and trace replays, largely due to the ease of use with which synthetic workloads can be employed. Given the popularity of synthetic benchmark workloads, we believe that an ideal benchmark for file and storage systems combines the *ease of use* of a synthetic workload with the *representativeness* of a real workload.

The process of creating such a benchmark should capture the essence of the original workload in a way that the synthetic workload remains representative; the process should be reproducible so that others can verify the authenticity of the benchmark workload without having to rely on the interpretation of the benchmark writer; and finally, creating synthetic workloads should be practical and encourage developers to adopt complex, real workloads into mainstream benchmarking.

While creating representative benchmark workloads is not an entirely solved problem, significant steps have been taken towards this goal. Empirical studies of file-system access patterns [17, 58, 100] and file-system activity traces [113, 133] have led to work on synthetic workload generators [12, 44] and methods for trace replay [14, 85]. However, automated workload synthesizers are hard to write. Current methods for creating synthetic benchmark workloads are largely based on the benchmark writer's interpretation of the real workload, and how it exercises the system API. This is insufficient since even a simple operation through the API may end up exercising the file system in very different ways due to effects of features such as caching and prefetching.

Determining whether or not two workloads stress a system in the same way is a challenging question; certainly, the domain of the system under test has a large impact on which features of the two workloads must be identical for the resulting performance to be identical. We believe that in order to create an equivalent synthetic workload for file and storage systems, one must mimic not the system calls, but the *function calls* exercised during workload execution, in order to be *functionally equivalent*. It is our hypothesis that if two workloads execute roughly the same set of function calls within the file system, that they will be roughly equivalent to one another.

Towards this end, we develop a tool called CodeMRI (an "MRI" for code if you will), that leverages file-system domain knowledge and a small amount of system profiling in order to better understand how a workload is stressing the system, and eventually construct a synthetic equivalent. Our initial experience with CodeMRI has shown promise in deconstructing real workloads; we believe it can provide essential building blocks towards developing automated workload synthesizers that are representative of real workloads.

1.4 Overview

The rest of the dissertation is organized as follows.

- **Representative and Reproducible Benchmarking State:** Chapter 2 presents our five-year study of file-system metadata; we analyze both static and longitudinal properties of file-system metadata that are representative of Windows PC systems in a corporate environment; this study is used as an exemplar for representative metadata properties throughout the rest of the dissertation.

  Chapter 3 presents our design, implementation and evaluation of Impressions, a framework to generate statistically accurate file-system images with realistic metadata and content. Impressions is flexible, supporting user-specified constraints on various file-system parameters using a number of statistical techniques to generate consistent, reproducible images for benchmarking.

- **Practical Large-Scale Benchmarking:** Chapter 4 presents our design, implementation and evaluation of Compressions, a practical scale-down system for running large, complex benchmarks. Compressions makes it feasible to run workloads that were otherwise infeasible to run with modest storage infrastructure, while also reducing the time taken to run the benchmark.

- **Representative, Reproducible and Practical Benchmark Workloads:**

  We discuss file-system benchmark workloads in Chapter 5; herein, we first present the requirements for generating representative synthetic benchmarks, then describe our initial successes with deconstructing real workloads, and finally discuss challenges in perfecting an automated benchmark workload synthesizer.

- **Conclusions and Future Work:** Chapter 6 concludes this dissertation, first summarizing the contributions of our work and discussing the lessons learned, and second, outlining possible avenues for future research that open up from this dissertation.

Chapter 2

Five Year Study of File System Metadata

Detailed knowledge of file-system metadata is essential for designing and benchmarking file and storage systems. Real world information about the structure, organization and contents of files and directories stored in file systems is crucial both to make informed design decisions and to accurately evaluate these systems. However useful, real world data is hard to collect and analyze, forcing storage system designers to rely on anecdotal information and often outdated rules of thumb.

In this chapter we present the first large-scale longitudinal study of file-system metadata. To perform this study, every year from 2000 to 2004, we collected snapshots of metadata from over ten thousand file systems on Windows desktop computers at Microsoft Corporation. We gathered this data by mass-emailing a scanning program to Microsoft's employees, and we had a 22% participation rate every year. Our resulting datasets contain metadata from 63,398 distinct file systems, 6457 of which provided snapshots in multiple years.

This project was a longitudinal extension of an earlier study performed by the Microsoft co-authors in 1998 [41], which was an order of magnitude larger than any prior study of file-system metadata. The earlier study involved a single capture of file-system metadata, and it focused on lateral variation among file systems at a moment in time. By contrast, the present study focuses on longitudinal changes in file systems over a five-year time span.

In particular, we study temporal changes in the size, age, and type frequency of files; the size of directories; the structure of the file-system namespace; and various characteristics of file systems, including file and directory population, storage capacity, storage consumption, and degree of file modification.

The contributions of this work are threefold. First, we contribute the collected data set, which we have sanitized and made available for general use [7]. This is the largest set of file-system metadata ever collected, and it spans the longest time period of any sizeable metadata collection. Second, we contribute all of our research observations, including:

- The space used in file systems has increased over the course of our study from year 2000 to 2004, not only because mean file size has increased (from 108 KB to 189 KB), but also because the number of files has increased (from 30K to 90K).
- Eight file-name extensions account for over 35% of files, and nine file-name extensions account for over 35% of the bytes in files. The same sets of extensions have remained popular for many years.
- The fraction of file-system content created or modified locally has decreased over time. In the first year of our study, the median file system had 30% of its files created or modified locally, and four years later this percentage was 22%.
- Directory size distribution has not notably changed over the years of our study. In each year, directories have had very few subdirectories and a modest number of entries. 90% of them have had two or fewer subdirectories, and 90% of them have had 20 or fewer total entries.
- The fraction of file system storage residing in the namespace subtree meant for user documents and settings has increased in every year of our study, starting at 7% and rising to 15%. The fraction residing in the subtree meant for system files has also risen over the course of our study, from 2% to 11%.

- File system capacity has increased dramatically during our study, with median capacity rising from 5 GB to 40 GB. One might expect this to cause drastic reductions in file system fullness, but instead the reduction in file system fullness has been modest. Median fullness has only decreased from 47% to 42%.
- Over the course of a single year, 80% of file systems become fuller and 18% become less full.

Third, we contribute a generative, probabilistic model for how directory trees are created. Our model explains the distribution of directories by depth in the namespace tree, and it also explains the distribution of the count of subdirectories per directory. This is the first generative model that characterizes the process by which file-system namespaces are constructed.

We believe that analysis of longitudinal file system data is of interest to many sets of people with diverse concerns about file system usage. For instance:

- developers of file systems, including desktop, server, and distributed file systems
- storage area network designers
- developers of file system utilities, such as backup, anti-virus, content indexing, encryption, and disk space usage visualization
- storage capacity planners
- disk manufacturers, especially those using gray-box techniques to enable visibility into the file system at the disk level [16]
- multitier storage system developers

Throughout this chapter, after discussing our findings and what we consider to be the most interesting summaries of these findings, we will present some examples of interesting implications for the people enumerated above.

The rest of this chapter is organized as follows. §2.1 describes the methodology of our data collection, analysis, and presentation. §2.2, §2.3, and §2.4 present our findings on, respectively, files, directories, and space usage. §2.5 surveys related work and compares our study with others that have been conducted in the past. §2.6 presents a discussion of our findings and §2.7 concludes the chapter.

2.1 Methodology

This section describes the methodology we applied to collecting, analyzing, and presenting the data.

2.1.1 Data collection

We developed a simple program that traverses the directory tree of each local, fixed-disk file system mounted on a computer. The program records a snapshot of all metadata associated with each file or directory, including hidden files and directories. This metadata includes name, size, timestamps (file or directory creation and modification), and attributes. The program also records the parent-child relationships of nodes in the namespace tree, as well as some system configuration information. The program records file names in an encrypted form. We used this program to collect information from NTFS and FAT based file systems. Apart from the file system information, the scanning program also records some system configuration such as volume and user ID, number of users on the computer, OS version and build, and information related to the processor on the machine.

We also wrote automated tools that decrypt the file names for computing aggregate statistics, but for privacy reasons we do not look at the decrypted file names directly, which places some limits on our analyses. In post-processing, we remove metadata relating to the system paging file, because this is part of the virtual memory system rather than the file system.

In the autumn of every year from 2000 to 2004, we distributed the scanning program via email to a large subset of the employees of Microsoft, with a request for the recipients to run the program on their desktop machines. The snapshots were collected entirely at Microsoft's primary campus in Redmond, WA. The sample population consists entirely of Microsoft Windows machines with users ranging from developers, system administrators, non-technical staff and researchers.

As an incentive to participate, we held a lottery in which each scanned machine counted as an entry, with a single prize of a night's stay at a nearby resort hotel. The specific subset of people we were permitted to poll varied from year to year based on a number of factors; however, despite variations in user population and in other distribution particulars, we observed approximately a 22% participation rate every year.

We scanned desktops rather than servers because at Microsoft, files are typically stored on individual desktops rather than centralized servers. We collected the data via voluntary participation rather than random selection because the company only permitted the former approach; note that this voluntary approach may have produced selection bias.

2.1.2 Data properties

Table 2.1 itemizes some properties of each year's data collection. The primary collection period ran between the listed start and end dates, which mark the beginning of our emailing requests and the last eligible day for the lottery. Some snapshots continued to trickle in after the primary collection period; we used these in our analyses as well.

Table 2.2 itemizes the breakdown of each year's snapshots according to file-system type. 80% of our snapshots came from NTFS [136], the main file system for operating systems in the Windows NT family; 5% from FAT [89], a 16-bit file system dating from DOS; and 15% from FAT32 [89], a 32-bit upgrade of FAT developed for Windows 95.

Year	Period	Users	Machs	FSs
2000	13 Sep – 29 Sep	5396	6051	11,654
2001	8 Oct – 2 Nov	7539	9363	16,022
2002	30 Sep – 1 Nov	7158	9091	15,011
2003	13 Oct – 14 Nov	7436	9262	14,633
2004	5 Oct – 12 Nov	7180	8729	13,505

Table 2.1 Properties of each year's dataset

For some analyses, we needed a way to establish whether two file-system snapshots from different years refer to the same file system. "Sameness" is not actually a well-formed notion; for example, it is not clear whether a file system is still the same after its volume is extended. We defined two snapshots to refer to the same file system if and only if they have the same user name, computer name, volume ID, drive letter, and total space. The need for some of these conditions was not obvious at first. For example, we added drive letter because some drives on some machines are multiply mapped, and we added total space so that a volume set would not be considered the same if a new volume were added to the set. Based on this definition, Table 2.3 shows the number of snapshots for which we have consecutive-year information.

2.1.3 Data presentation

Many of our graphs have horizontal axes that span a large range of non-negative numbers. To represent these ranges compactly, we use a logarithmic scale for non-zero values, but we also include an abscissa for the zero value, even though zero does not strictly belong on a logarithmic scale.

We plot most histograms with line graphs rather than bar graphs because, with five or more datasets on a single plot, bar graphs can become difficult to read. For each bin in the

Year	NTFS	FAT32	FAT	Other	Total
2000	7,015	2,696	1,943	0	11,654
2001	11,791	3,314	915	2	16,022
2002	12,302	2,280	429	0	15,011
2003	12,853	1,478	302	0	14,633
2004	12,364	876	264	1	13,505
Total	56,325	10,644	3,853	3	70,825

Table 2.2 File system types in datasets

Start	1	2	3	4	5
2000	11,654	950	234	63	18
2001	16,022	1,833	498	144	-
2002	15,011	1,852	588	-	-
2003	14,633	1,901	-	-	-
2004	13,505	-	-	-	-
Total	70,825	6,536	1,320	207	18

Table 2.3 **File systems with snapshots in successive years.** *Number of file systems for which we have snapshots in the n consecutive years starting with each year. The numbers in this table are cumulative for the n years. For instance, there are 1,852 file systems for which we have snapshots from both 2002 and 2003, and 588 file systems for which we have snapshots in the three years 2002, 2003 and 2004.*

histogram, we plot a point (x, y) where x is the midpoint of the bin and y is the size of the bin. We use the geometric midpoint when the x axis uses a logarithmic scale. We often plot un-normalized histograms rather than probability density functions (PDFs) for two reasons: First, the graphs expose more data if we do not normalize them. Second, because the count of files and directories per file system has grown substantially over time, not normalizing allows us to plot multiple years' curves on the same chart without overlapping to the point of unreadability.

Whenever we use the prefix K, as in KB, we mean 2^{10}. Similarly, we use M for 2^{20} and G for 2^{30}.

2.1.4 Limitations

All our data comes from a relatively homogeneous sample of machines: Microsoft desktops running Windows. Since past studies [116, 153] have shown that file system characteristics can vary from one environment to another, our conclusions may not be applicable to substantially different environments. For instance, our conclusions are likely not applicable to file system server workloads, and it is unclear to what extent they can be generalized to non-Windows operating systems. It may also be that artifacts of Microsoft policy, such as specific software distributions that are common or disallowed, may yield results that would not apply to other workloads.

2.2 Files

2.2.1 File count per file system

Figure 2.1 plots cumulative distribution functions (CDFs) of file systems by count of files. The count of files per file system has increased steadily over our five-year sample period: The arithmetic mean has grown from 30K to 90K files and the median has grown from 18K to 52K files.

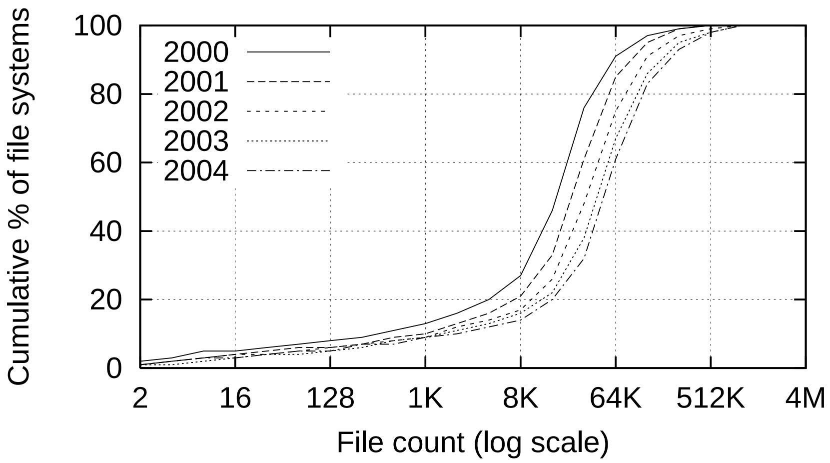


Figure 2.1 CDFs of file systems by file count

The count of files per file system is going up from year to year, and, as we will discuss in §2.3.1, the same holds for directories. Thus, file system designers should ensure their metadata tables scale to large file counts. Additionally, we can expect file system scans that examine data proportional to the number of files and/or directories to take progressively longer. Examples of such scans include virus scans and metadata integrity checks following block corruption. Thus, it will become increasingly useful to perform these checks efficiently, perhaps by scanning in an order that minimizes movement of the disk arm.

2.2.2 File size

This section describes our findings regarding file size. We report the size of actual content, ignoring the effects of internal fragmentation, file metadata, and any other overhead. We observe that the overall file size distribution has changed slightly over the five years of our study. By contrast, the majority of stored bytes are found in increasingly larger files. Moreover, the latter distribution increasingly exhibits a double mode, due mainly to database and blob (binary large object) files.

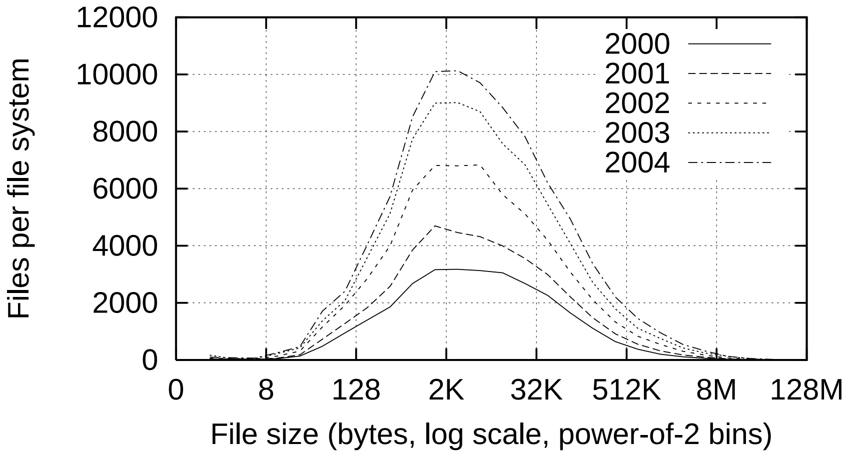


Figure 2.2 Histograms of files by size

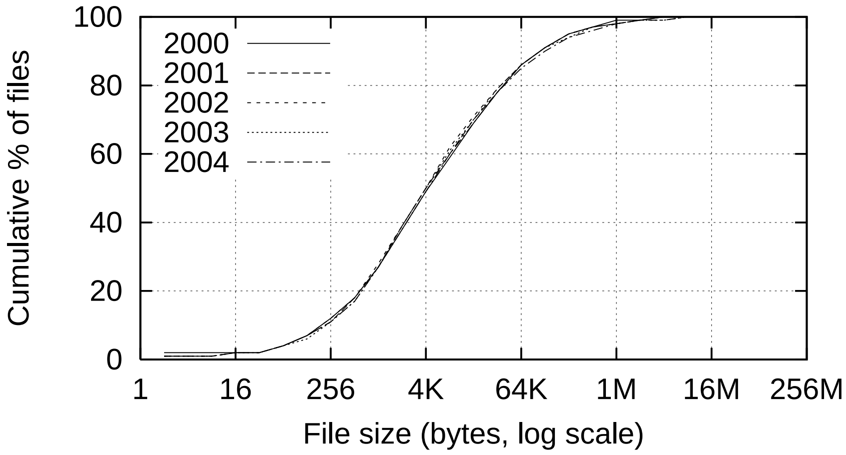


Figure 2.3 CDFs of files by size

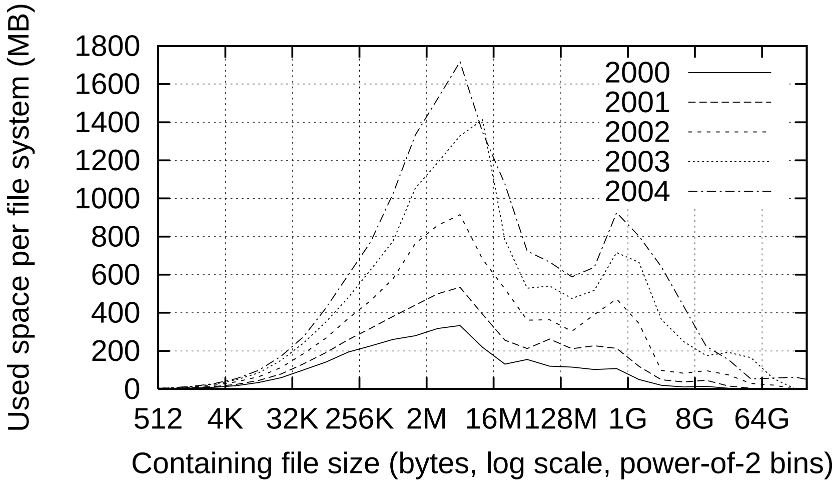


Figure 2.4 Histograms of bytes by containing file size

Figure 2.2 plots histograms of files by size and Figure 2.3 plots the corresponding CDFs. We see that the absolute count of files per file system has grown significantly over time, but the general shape of the distribution has not changed significantly. Although it is not visible on the graph, the arithmetic mean file size has grown by 75% from 108 KB to 189 KB. In each year, 1–1.5% of files have a size of zero.

The growth in mean file size from 108 KB to 189 KB over four years suggests that this metric grows roughly 15% per year. Another way to estimate this growth rate is to compare our 2000 result to the 1981 result of 13.4 KB obtained by Satyanarayanan [120]. This comparison estimates the annual growth rate as 12%. Note that this latter estimate is somewhat flawed, since it compares file sizes from two rather different environments.

Figure 2.4 plots histograms of bytes by containing file size, alternately described as histograms of files weighted by file size. Figure 2.5 plots CDFs of these distributions. We observe that the distribution of file size has shifted to the right over time, with the median weighted file size increasing from 3 MB to 9 MB. Also, the distribution exhibits a double mode that has become progressively more pronounced. The corresponding distribution in

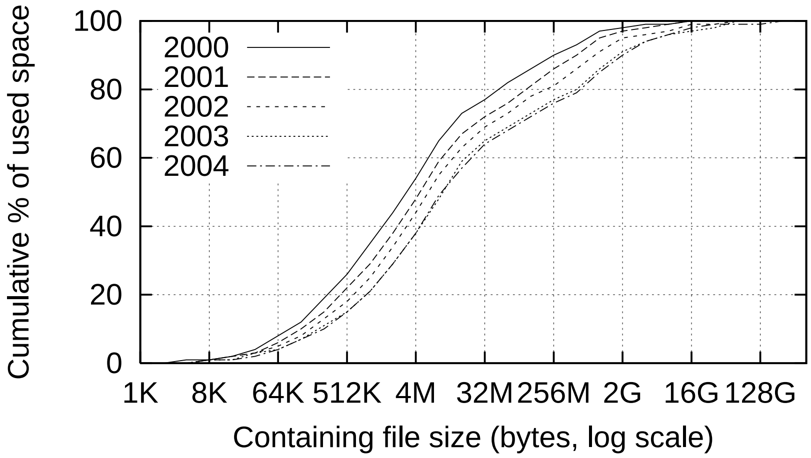


Figure 2.5 CDFs of bytes by containing file size

our 1998 study did not show a true second mode, but it did show an inflection point around 64 MB, which is near the local minimum in Figure 2.4.

To study this second peak, we broke out several categories of files according to filename extension. Figure 2.6 re-plots the 2004 data from Figure 2.4 as a stacked bar chart, with the contributions of video, database, and blob files indicated. We see that most of the bytes in large files are in video, database, and blob files, and that most of the video, database, and blob bytes are in large files.

Our finding that different types of files have different size distributions echoes the findings of other studies. In 1981, Satyanarayanan [120] found this to be the case on a shared file server in an academic environment. In 2001, Evans and Kuenning also noted this phenomenon in their analysis of 22 machines running various operating systems at Harvey Mudd College and Marine Biological Laboratories [46]. The fact that this finding is consistent across various different environments and times suggests that it is fundamental.

There are several implications of the fact that a large number of small files account for a small fraction of disk usage, such as the following. First, it may not take much space to co-locate many of these files with their metadata. This may be a reasonable way to reduce

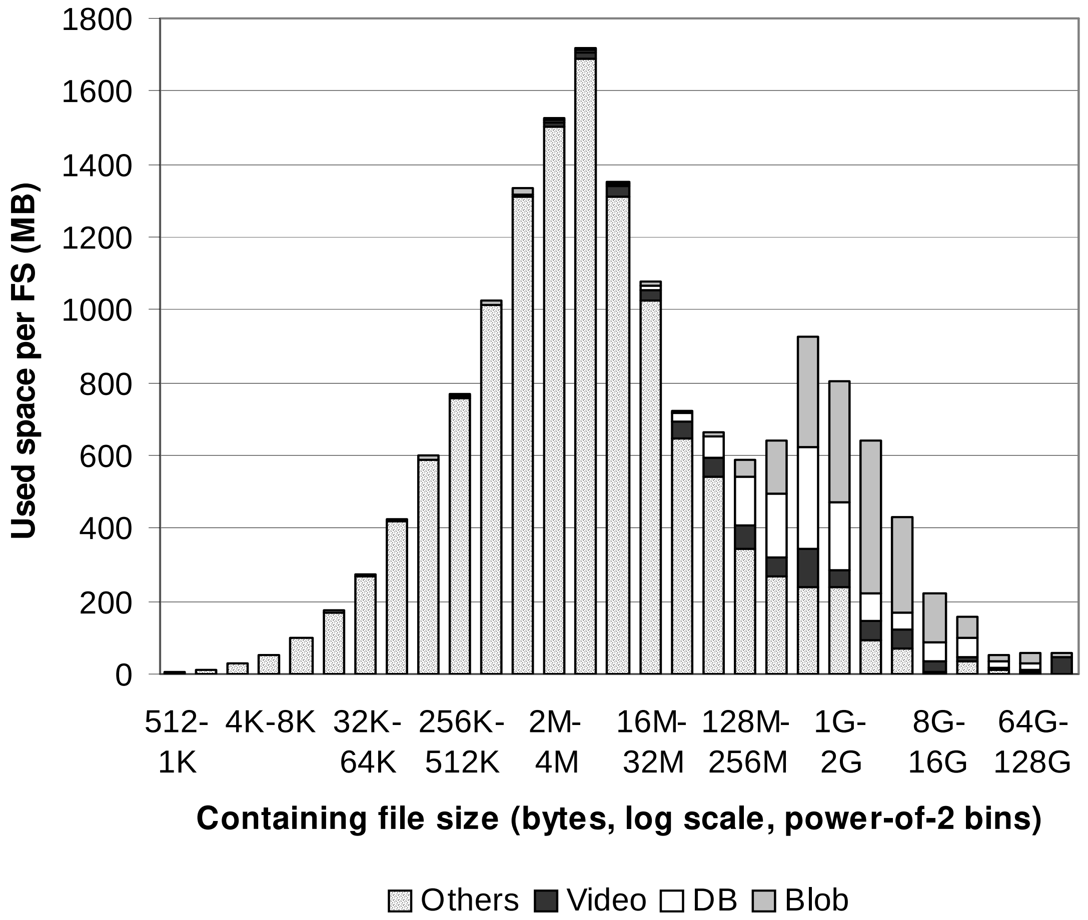


Figure 2.6 **Contribution of file types to Figure 2.4 for year 2004.** *Video means files with extension* `avi`*,* `dps`*,* `mpeg`*,* `mpg`*,* `vob`*, or* `wmv`*; DB means files with extension* `ldf`*,* `mad`*,* `mdf`*,* `ndf`*,* `ost`*, or* `pst`*; and Blob means files named* `hiberfil.sys` *and files with extension* `bak`*,* `bkf`*,* `bkp`*,* `dmp`*,* `gho`*,* `iso`*,* `pqi`*,* `rbf`*, or* `vhd`*.*

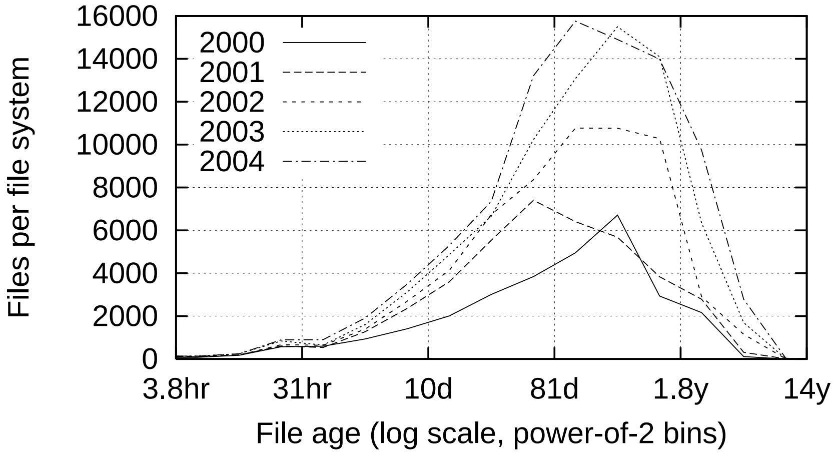


Figure 2.7 Histograms of files by age

the disk seek time needed to access these files. Second, a file system that co-locates several files in a single block, like ReiserFS [110], will have many opportunities to do so. This will save substantial space by eliminating internal fragmentation, especially if a large block size is used to improve performance. Third, designers of disk usage visualization utilities may want to show not only directories but also the names of certain large files.

2.2.3 File age

This subsection describes our findings regarding file age. Because file timestamps can be modified by application programs [87], our conclusions should be regarded cautiously.

Figure 2.7 plots histograms of files by age, calculated as the elapsed time since the file was created or last modified, relative to the time of the snapshot; we use the most recent timestamp from amongst the creation and modification timestamps to calculate file age. Figure 2.8 shows CDFs of this same data. The median file age ranges between 80 and 160 days across datasets, with no clear trend over time.

The distribution of file age is not memoryless, so the age of a file is useful in predicting its remaining lifetime. So, systems such as archival backup systems can use this distribution

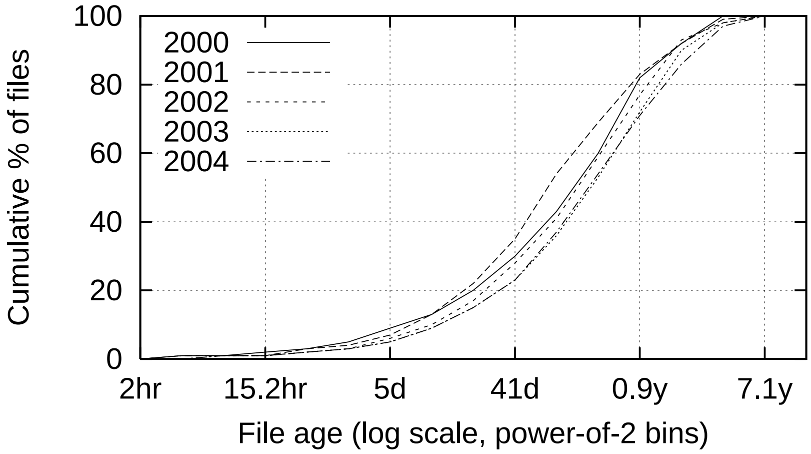


Figure 2.8 CDFs of files by age

to make predictions of how much longer a file will be needed based on how old it is. Since the distribution of file age has not appreciably changed across the years, we can expect that a prediction algorithm developed today based on the latest distribution will apply for several years to come.

2.2.4 File-name extensions

This subsection describes our findings regarding popular file types, as determined by file-name extension. Although the top few extensions have not changed dramatically over our five-year sample period, there has been some change, reflecting a decline in the relative prevalence of web content and an increase in use of virtual machines. The top few extensions account for nearly half of all files and bytes in file systems.

In old DOS systems with 8.3-style file names, the extension was the zero to three characters following the single dot in the file name. Although Windows systems allow file names of nearly arbitrary length and containing multiple dots, many applications continue to indicate their file types by means of extensions. For our analyses, we define an extension as the five-or-fewer characters following the last dot in a file name. If a name has no dots or

Extension	Typical Usage
cpp	C++ source code
dll	Dynamic link library
exe	Executable
gif	Image in Graphic Interchange Format
h	Source code header
htm	File in hypertext markup language
jpg	Image in JPEG format
lib	Code library
mp3	Music file in MPEG Layer III format
pch	Precompiled header
pdb	Source symbols for debugging
pst	Outlook personal folder
txt	Text
vhd	Virtual hard drive for virtual machine
wma	Windows Media Audio

Table 2.4 Typical usage of popular file extensions

has more than five characters after the last dot, we consider that name to have no extension, which we represent with the symbol Ø. As a special case, if a file name ends in `.gz`, `.bz2`, and `.Z`, then we ignore that suffix when determining extension. We do this because these are types of compressed files wherein the actual content type is indicated by the characters prior to the compression extension. To understand the typical usage of the file extensions we discuss in this section, see Table 2.4.

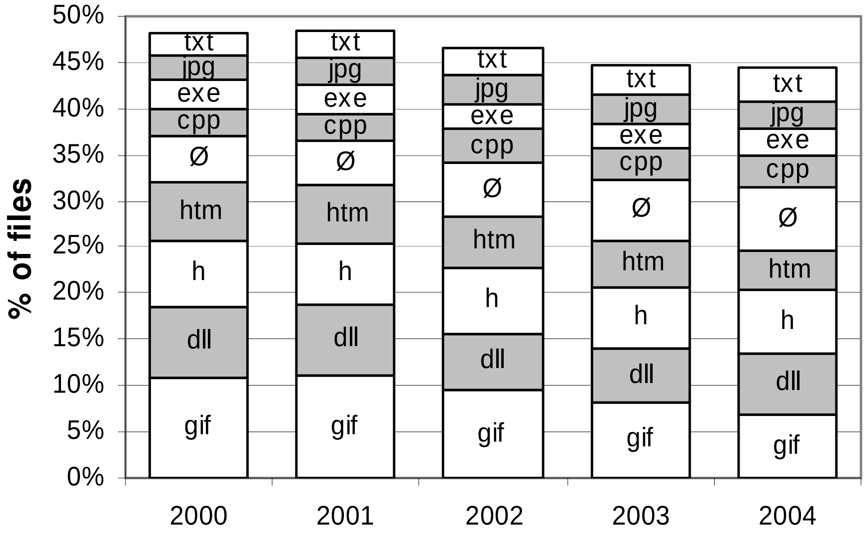


Figure 2.9 Fraction of files with popular extensions

Figure 2.9 plots, for the nine extensions that are the most popular in terms of file count, the fraction of files with that extension. The fractions are plotted longitudinally over our five-year sample period. The most notable thing we observe is that these extensions' popularity is relatively stable—the top five extensions have remained the top five for this entire time. However, the relative popularity of `gif` files and `htm` files has gone down steadily since 2001, suggesting a decline in the popularity of web content relative to other ways to fill one's file system.

Figure 2.10 plots, for the ten extensions that are the most popular in terms of summed file size, the fraction of file bytes residing in files with that extension. Across all years, dynamic link libraries (`dll` files) contain more bytes than any other file type. Extension `vhd`, which is used for virtual hard drives, is consuming a rapidly increasing fraction of file-system space, suggesting that virtual machine use is increasing. The null extension exhibits a notable anomaly in 2003, but we cannot investigate the cause without decrypting the file names in our datasets, which would violate our privacy policy.

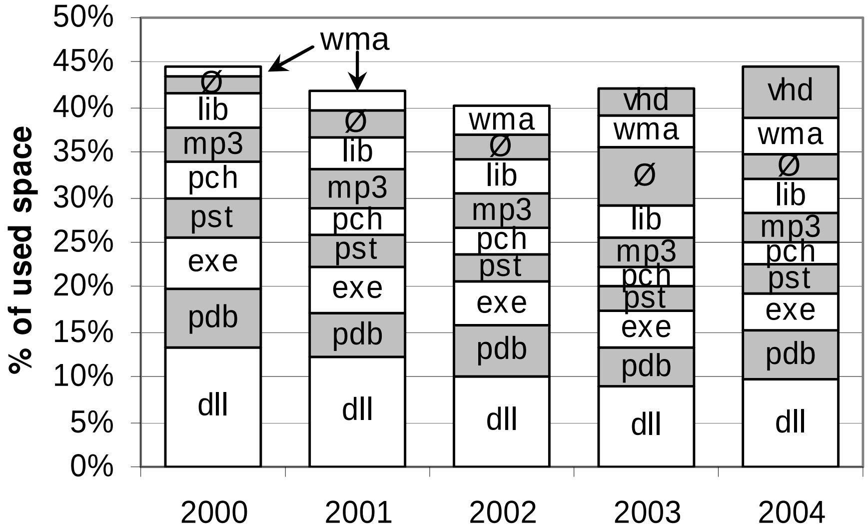


Figure 2.10 Fraction of bytes in files with popular extensions

Since files with the same extension have similar properties and requirements, some file system management policies can be improved by including special-case treatment for particular extensions. Such special-case treatment can be built into the file system or autonomically and dynamically learned [84]. Since nearly half the files, and nearly half the bytes, belong to files with a few popular extensions, developing such special-case treatment for only a few particular extensions can optimize performance for a large fraction of the file system. Furthermore, since the same extensions continue to be popular year after year, one can develop special-case treatments for today's popular extensions and expect that they will still be useful years from now.

2.2.5 Unwritten files

Figures 2.11 and 2.12 plot histograms and CDFs, respectively, of file systems by percentage of files that have not been written since they were copied onto the file system. We identify such files as ones whose modification timestamps are earlier than their creation timestamps, since the creation timestamp of a copied file is set to the time at which the

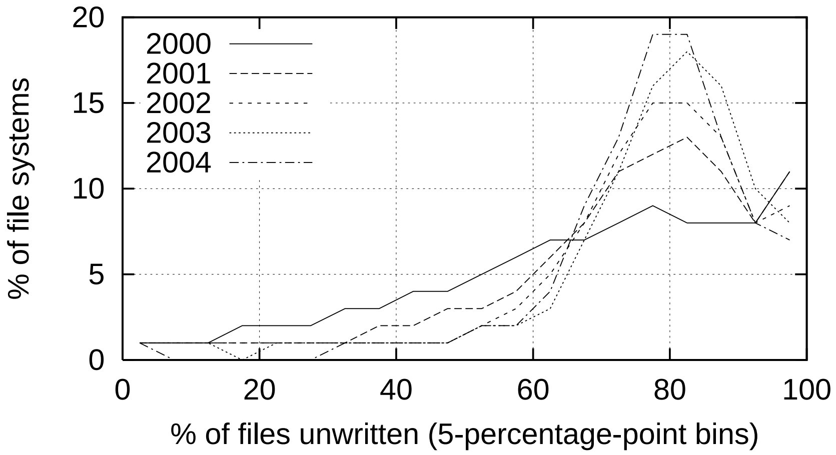


Figure 2.11 Histograms of file systems by percentage of files unwritten

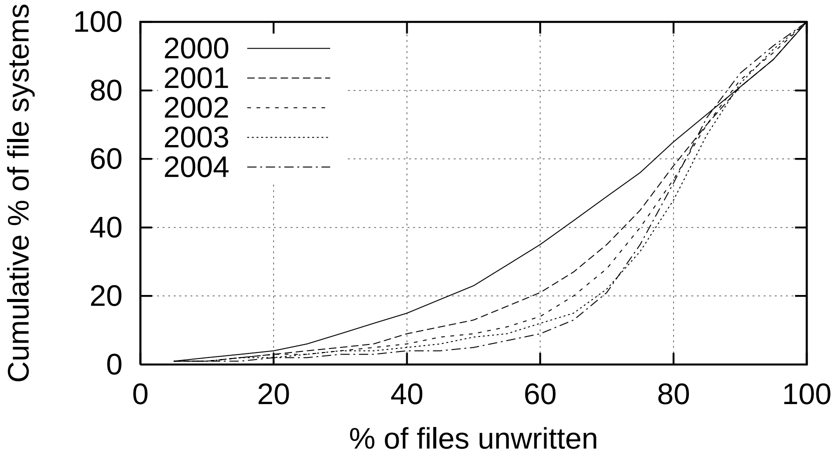


Figure 2.12 CDFs of file systems by percentage of files unwritten

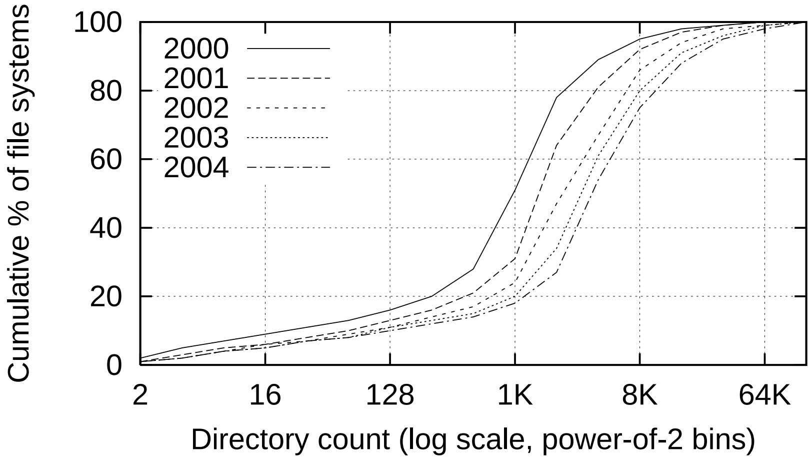


Figure 2.13 CDFs of file systems by directory count

copy was made, but its modification timestamp is copied from the original file. Over our sample period, the arithmetic mean of the percentage of locally unwritten files has grown from 66% to 76%, and the median has grown from 70% to 78%. This suggests that users locally contribute to a decreasing fraction of their systems' content. This may in part be due to the increasing amount of total content over time.

Since more and more files are being copied across file systems rather than generated locally, we can expect identifying and coalescing identical copies to become increasingly important in systems that aggregate file systems. Examples of systems with such support are the FARSITE distributed file system [2], the Pastiche peer-to-peer backup system [35], and the Single Instance Store in Windows file servers [23].

2.3 Directories

2.3.1 Directory count per file system

Figure 2.13 plots CDFs of file systems by count of directories. The count of directories per file system has increased steadily over our five-year sample period: The arithmetic mean has grown from 2400 to 8900 directories and the median has grown from 1K to 4K directories.

We discussed implications of the rising number of directories per file system earlier. The count of and directories per file system is going up from year to year, and, as we discussed in §2.2.1, the same holds for files. Thus, file system designers should ensure their metadata tables scale not only to large file counts, but also for large directory counts. Additionally, we can expect file system scans that examine data proportional to the number of files and/or directories to take progressively longer. Examples of such scans include virus scans and metadata integrity checks following block corruption. Thus, it will become increasingly useful to perform these checks efficiently. One example is the XFS file system [141] that promises to deliver fast response times, even for directories with tens of thousands of entries.

2.3.2 Directory size

This section describes our findings regarding directory size, measured by count of contained files, count of contained subdirectories, and total entry count. None of these size distributions has changed appreciably over our sample period, but the mean count of files per directory has decreased slightly.

Figure 2.14 plots CDFs of directories by size, as measured by count of files in the directory. It shows that although the absolute count of directories per file system has grown significantly over time, the distribution has not changed appreciably. Across all years, 23–25% of directories contain no files, which marks a change from 1998, in which only 18% contained no files and there were more directories containing one file than those containing none. The arithmetic mean directory size has decreased slightly and steadily from 12.5 to 10.2 over the sample period, but the median directory size has remained steady at 2 files.

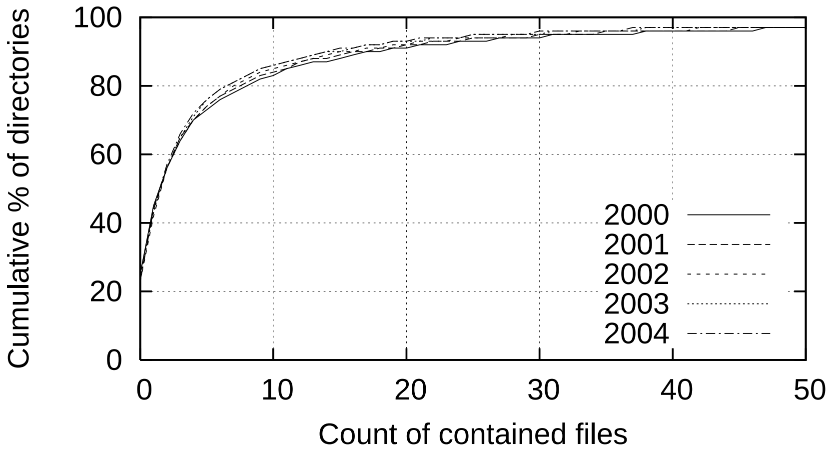


Figure 2.14 CDFs of directories by file count

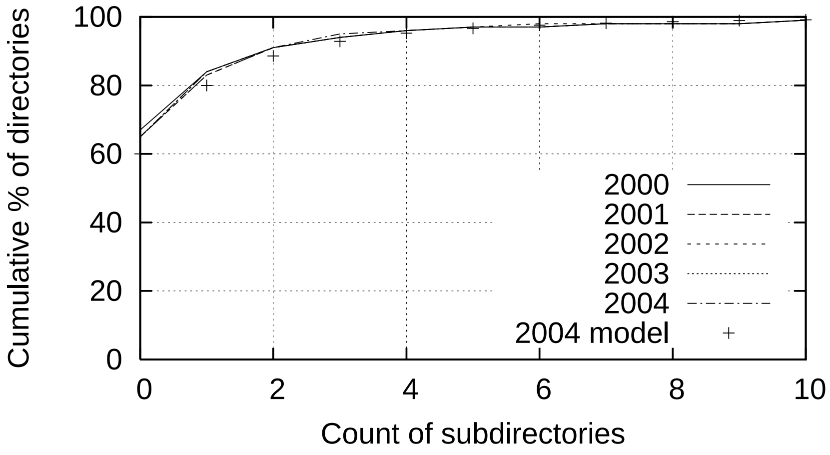


Figure 2.15 CDFs of directories by subdirectory count

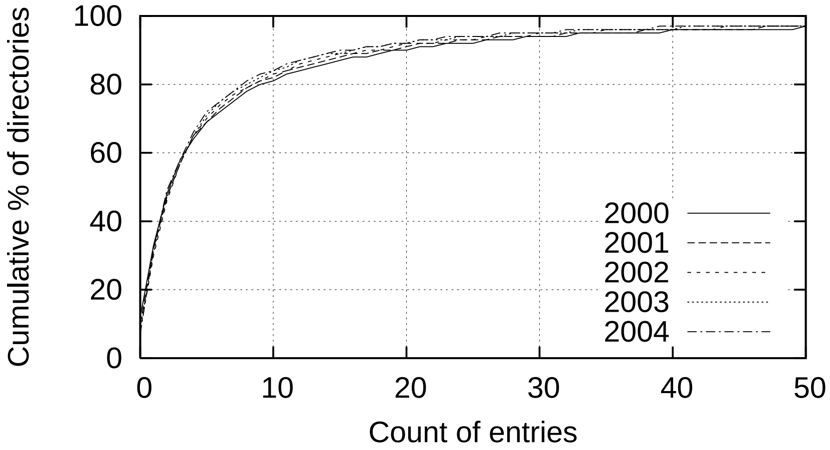


Figure 2.16 CDFs of directories by entry count

Figure 2.15 plots CDFs of directories by size, as measured by count of subdirectories in the directory. It includes a model approximation we will discuss later in §2.3.5. This distribution has remained unchanged over our sample period. Across all years, 65–67% of directories contain no subdirectories, which is similar to the 69% found in 1998.

Figure 2.16 plots CDFs of directories by size, as measured by count of total entries in the directory. This distribution has remained largely unchanged over our sample period. Across all years, 46–49% of directories contain two or fewer entries.

Since there are so many directories with a small number of files, it would not take much space to co-locate the metadata for most of those files with those directories. Such a layout would reduce seeks associated with file accesses. Therefore, it might be useful to preallocate a small amount of space near a new directory to hold a modest amount of child metadata. Similarly, most directories contain fewer than twenty entries, suggesting using an on-disk structure for directories that optimizes for this common case.

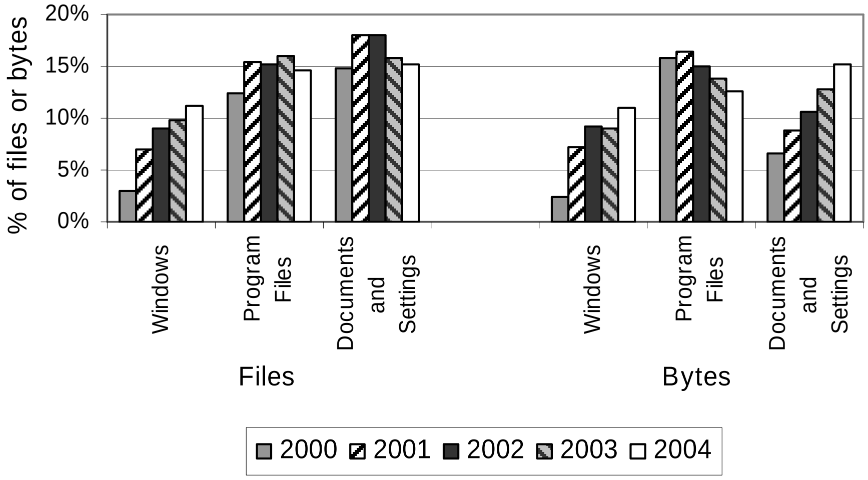


Figure 2.17 Fraction of files and bytes in special subtrees

2.3.3 Special directories

This section describes our findings regarding the usage of Windows special directories. We find that an increasing fraction of file-system storage is in the namespace subtree devoted to system files, and the same holds for the subtree devoted to user documents and settings.

Figure 2.17 plots the fraction of file-system files that reside within subtrees rooted in each of three special directories: `Windows`, `Program Files`, and `Documents and Settings`. This figure also plots the fraction of file-system bytes contained within each of these special subtrees.

For the `Windows` subtree, the fractions of files and bytes have both risen from 2–3% to 11% over our sample period, suggesting that an increasingly large fraction of file-system storage is devoted to system files. In particular, we note that Windows XP was released between the times of our 2000 and 2001 data collections.

For the `Program Files` subtree, the fractions of files and bytes have trended in opposite directions within the range of 12–16%. For the `Documents and Settings` subtree, the fraction of bytes has increased dramatically while the fraction of files has remained relatively stable.

The fraction of all files accounted for by these subtrees has risen from 25% to 40%, and the fraction of bytes therein has risen from 30% to 41%, suggesting that application writers and end users have increasingly adopted Windows' prescriptive namespace organization [29].

Backup software generally does not have to back up system files, since they are static and easily restored. Since system files are accounting for a larger and larger fraction of used space, it is becoming more and more useful for backup software to exclude these files.

On the other hand, files in the Documents and Settings folder tend to be the most important files to back up, since they contain user-generated content and configuration information. Since the percentage of bytes devoted to these files is increasing, backup capacity planners should expect, surprisingly, that their capacity requirements will increase *faster* than disk capacity is planned to grow. On the other hand, the percentage of files is not increasing, so they need not expect metadata storage requirements to scale faster than disk capacity. This may be relevant if metadata is backed up in a separate repository from the data, as done by systems such as EMC Centera [61].

2.3.4 Namespace tree depth

This section describes our findings regarding the depth of directories, files, and bytes in the namespace tree. We find that there are many files deep in the namespace tree, especially at depth 7. Also, we find that files deeper in the namespace tree tend to be orders-of-magnitude smaller than shallower files.

Figure 2.18 plots histograms of directories by their depth in the namespace tree, and Figure 2.19 plots CDFs of this same data; it also includes a model approximation we will discuss later in §2.3.5. The general shape of the distribution has remained consistent over

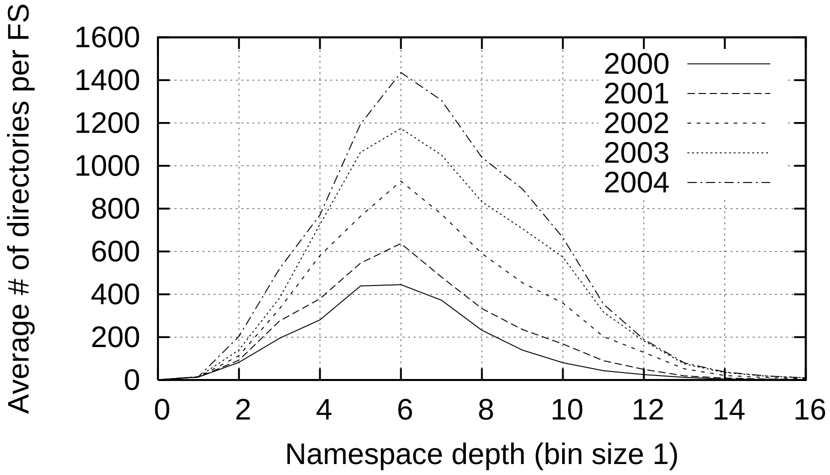


Figure 2.18 Histograms of directories by namespace depth

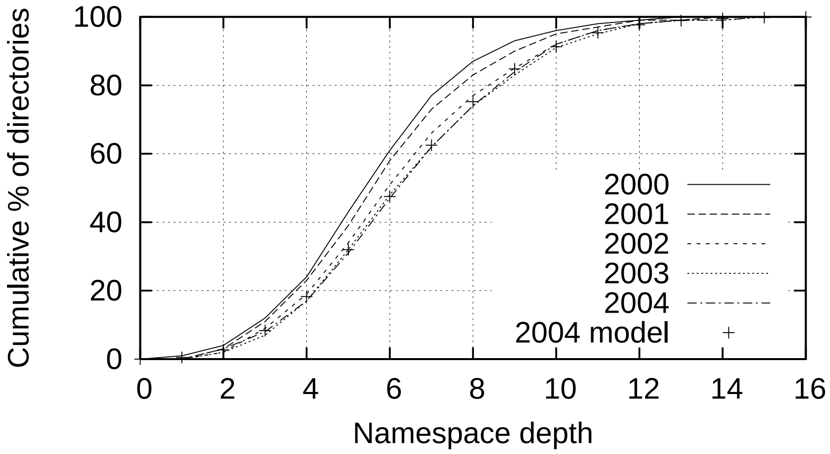


Figure 2.19 CDFs of directories by namespace depth

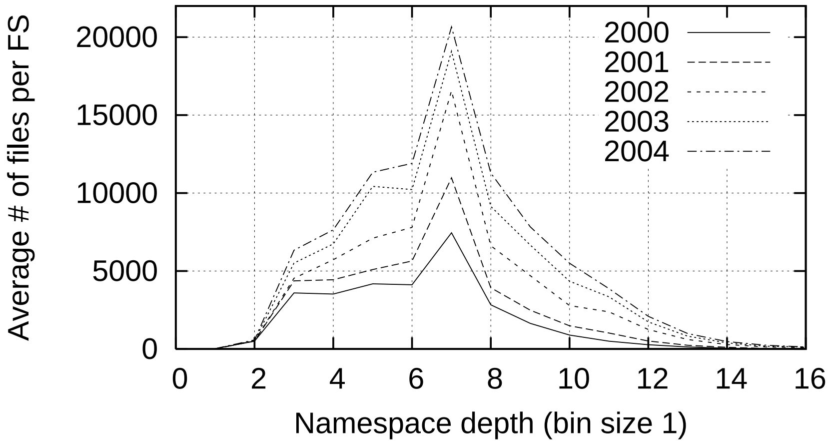


Figure 2.20 Histograms of files by namespace depth

our sample period, but the arithmetic mean has grown from 6.1 to 6.9, and the median directory depth has increased from 5 to 6.

Figure 2.20 plots histograms of file count by depth in the namespace tree, and Figure 2.21 plots CDFs of this same data. With a few exceptions, such as at depths 2, 3, and 7, these distributions roughly track the observed distributions of directory depth, indicating that the count of files per directory is mostly independent of directory depth. To study this more directly, Figure 2.22 plots the mean count of files per directory versus directory depth. There is a slight downward trend in this ratio with increasing depth, punctuated by three depths whose directories have greater-than-typical counts of files: at depth 2 are files in the `Windows` and `Program Files` directories; at depth 3 are files in the `System` and `System32` directories; and at depth 7 are files in the web cache directories.

Figure 2.23 replots the 2004 data from Figure 2.20 as a stacked bar chart, with the indicated contributions of the special namespace subtrees defined in the previous section. Absent these specific types, the distribution of file count by depth in the namespace tree is closely approximated by a Poisson distribution with $\lambda = 6.5$, as shown, yielding an MDCC of 1%. Figure 2.24 plots histograms of bytes by the depth of their containing files

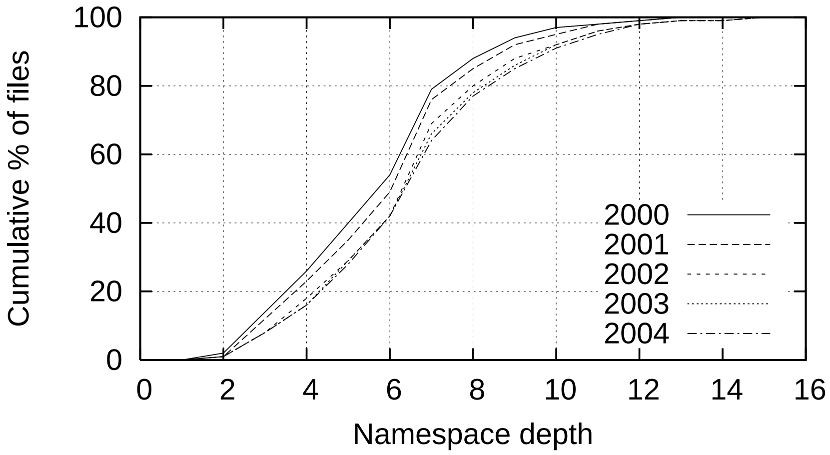


Figure 2.21 CDFs of files by namespace depth

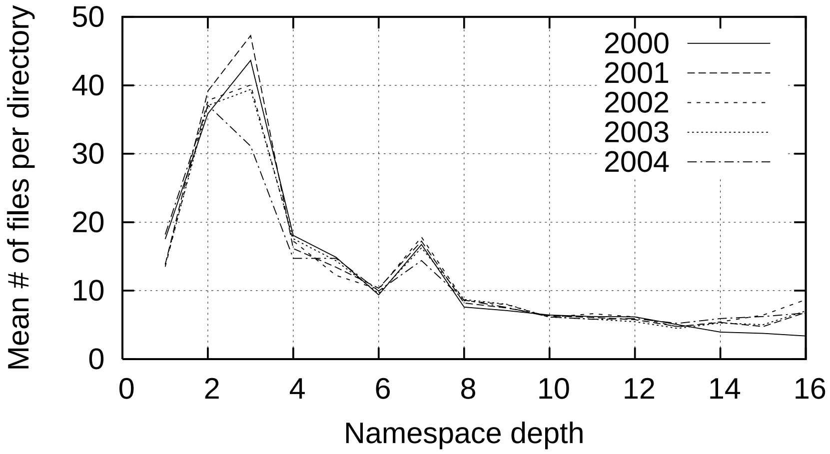


Figure 2.22 Files per directory vs. namespace depth

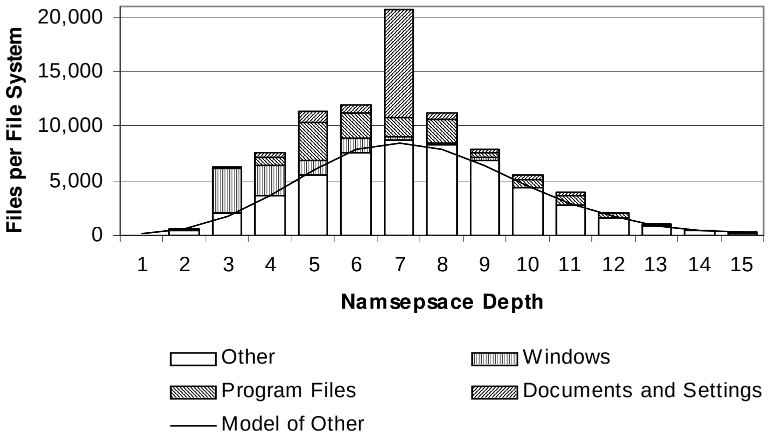


Figure 2.23 Contribution of special subtrees to histogram of 2004 files by namespace depth

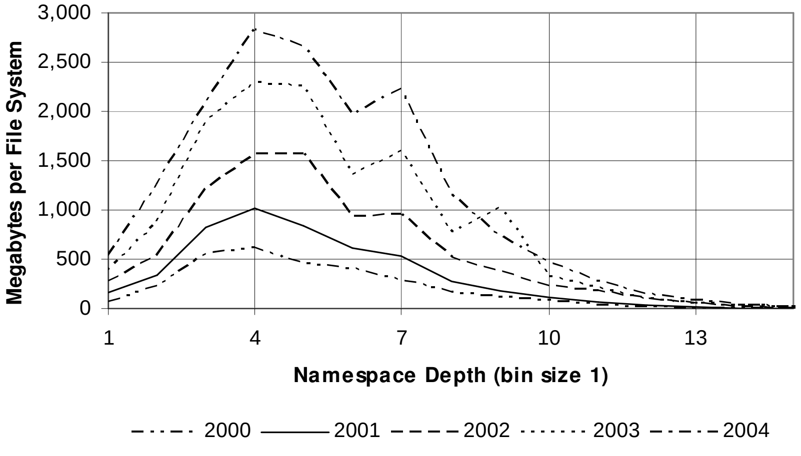


Figure 2.24 Histograms of bytes by namespace depth

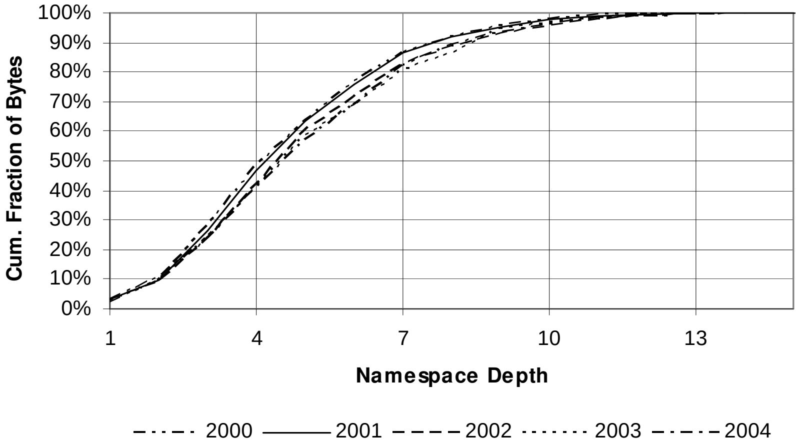


Figure 2.25 CDFs of bytes by namespace depth

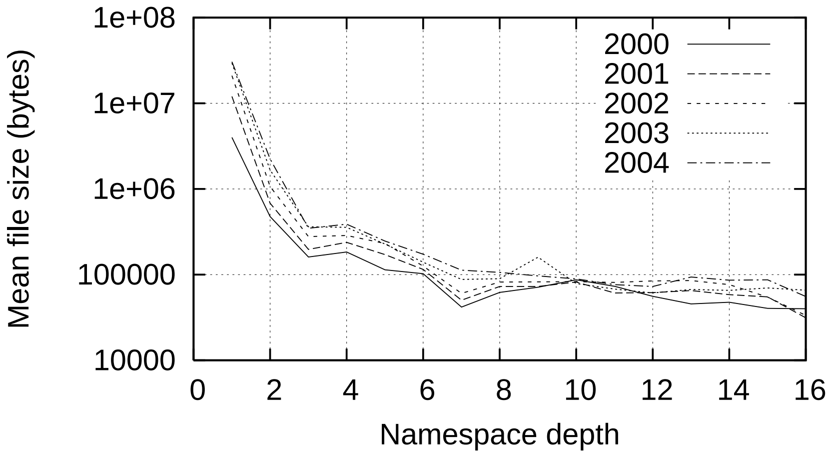


Figure 2.26 File size vs. namespace depth

in the namespace tree, and Figure 2.21 plots CDFs of this same data. These distributions do not closely track the observed distributions of file depth. In particular, files deeper in the namespace tree tend to be smaller than shallower ones.

This trend is more obvious in Figure 2.26, which plots the mean file size versus directory depth on a logarithmic scale. We see here that files deeper in the namespace tree tend to be smaller. The mean file size drops by two orders of magnitude between depth 1 and depth 3, and there is a drop of roughly 10% per depth level thereafter. This phenomenon occurs because most bytes are concentrated in a small number of large files (see Figures 2.2 and 2.4), and these files tend to reside in shallow levels of the namespace tree. In particular, the hibernation image file is located in the root.

Since many files and directories are deep in the namespace tree, efficient path lookup of deep paths should be a priority for file system designers. For instance, in distributed file systems where different servers are responsible for different parts of the namespace tree [2], deep path lookup may be expensive if not optimized. The high depth of many entries in the namespace may also be of interest to designers of file system visualization GUIs, to determine how much column space to allot for directory traversal. Furthermore, since the fraction of files at high depths is increasing across the years of our study, these lessons will become more and more important as years pass.

The clear trend of decreasing file size with increasing namespace tree depth suggests a simple coarse mechanism to predict future file size at time of file creation. File systems might use such prediction to decide where on disk to place a new file.

2.3.5 Namespace depth model

We have developed a generative model that accounts for the distribution of directory depth. The model posits that new subdirectories are created inside an existing directory in offset proportion to the count of subdirectories already in that directory.

In the study of file-system metadata by Douceur and Bolosky [41], they observed that the distribution of directories by depth could be approximated by a Poisson distribution with $\lambda = 4.38$, yielding a maximum displacement of cumulative curves (MDCC) of 2%. Poisson is also an acceptable approximation for the five datasets in the present study, with λ growing from 6.03 to 6.88 over the sample period, yielding MDCCs that range from 1% to 4%. However, the Poisson distribution does not provide an explanation for the behavior; it merely provides a means to approximate the result. By contrast, we have developed a generative model that accounts for the distribution of directory depths we have observed, with accuracy comparable to the Poisson model.

The generative model is as follows. A file system begins with an empty root directory. Directories are added to the file system one at a time. For each new directory, a parent directory is selected probabilistically, based on the count of subdirectories the parent currently has. Specifically, the probability of choosing each extant directory as a parent is proportional to $c(d) + 2$, where $c(d)$ is the count of extant subdirectories of directory d. We used Monte Carlo simulation to compute directory depth distributions according to this generative model. Given a count of directories in a file system, the model produces a distribution of directory depths that matches the observed distribution for file systems of that size. Figure 2.19 plots the aggregate result of the model for all file systems in the 2004 dataset. The model closely matches the CDF of observed directory depths, with an MDCC of 1%.

Our generative model accounts not only for the distribution of directory depth but also for that of subdirectory size. Figure 2.15 shows this for the 2004 dataset. The model closely matches the CDF, with an MDCC of 5%.

Intuitively, the proportional probability $c(d) + 2$ can be interpreted as follows: If a directory already has some subdirectories, it has demonstrated that it is a useful location for subdirectories, and so it is a likely place for more subdirectories to be created. The

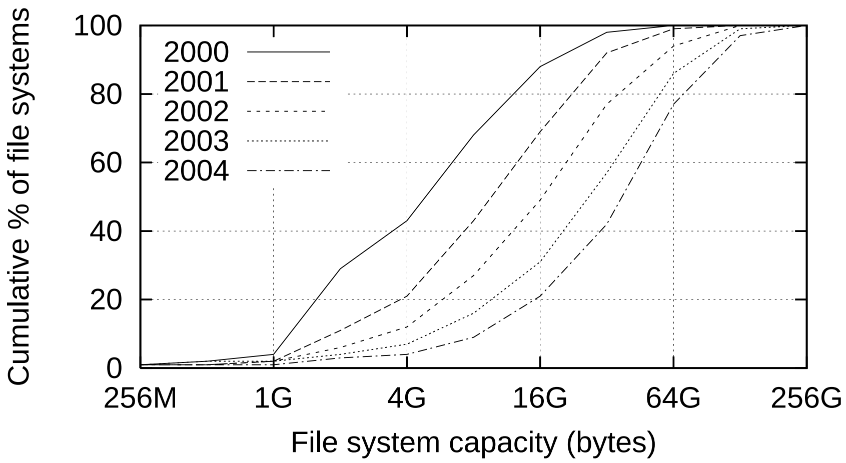


Figure 2.27 CDFs of file systems by storage capacity

more subdirectories it has, the more demonstrably useful it has been as a subdirectory home, so the more likely it is to continue to spawn new subdirectories. If the probability were proportional to $c(d)$ without any offset, then an empty directory could never become non-empty, so some offset is necessary. We found an offset of 2 to match our observed distributions very closely for all five years of our collected data, but we do not understand why the particular value of 2 should be appropriate. One hypothesis is that the value of 2 comes from the two default directories "." and ".." that are present in all directories.

2.4 Space Usage

2.4.1 Capacity and usage

Figure 2.27 plots CDFs of file system volumes by storage capacity, which has increased dramatically over our five-year sample period: The arithmetic mean has grown from 8 GB to 46 GB and the median has grown from 5 GB to 40 GB. The number of small-capacity file system volumes has dropped dramatically: Systems of 4 GB or less have gone from 43% to 4% of all file systems.

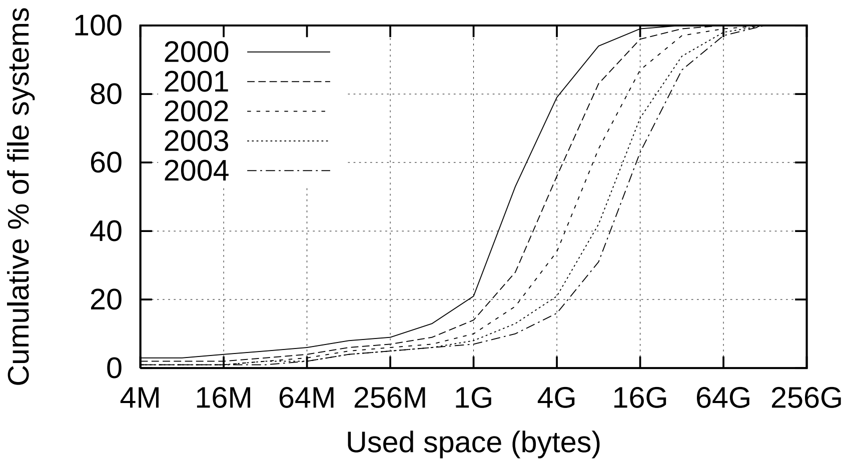


Figure 2.28 CDFs of file systems by total consumed space

Figure 2.28 plots CDFs of file systems by total consumed space, including not only file content but also space consumed by internal fragmentation, file metadata, and the system paging file. Space consumption increased steadily over our five-year sample period: The geometric mean has grown from 1 GB to 9 GB, the arithmetic mean has grown from 3 GB to 18 GB, and the median has grown from 2 GB to 13 GB.

Figure 2.29 plots CDFs of file systems by percentage of fullness, meaning the consumed space relative to capacity. The distribution is very nearly uniform for all years, as it was in our 1998 study. The mean fullness has dropped slightly from 49% to 45%, and the median file system has gone from 47% full to 42% full. By contrast, the aggregate fullness of our sample population, computed as total consumed space divided by total file-system capacity, has held steady at 41% over all years.

In any given year, the range of file system capacities in this organization is quite large. This means that software must be able to accommodate a wide range of capacities simultaneously existing within an organization. For instance, a peer-to-peer backup system must

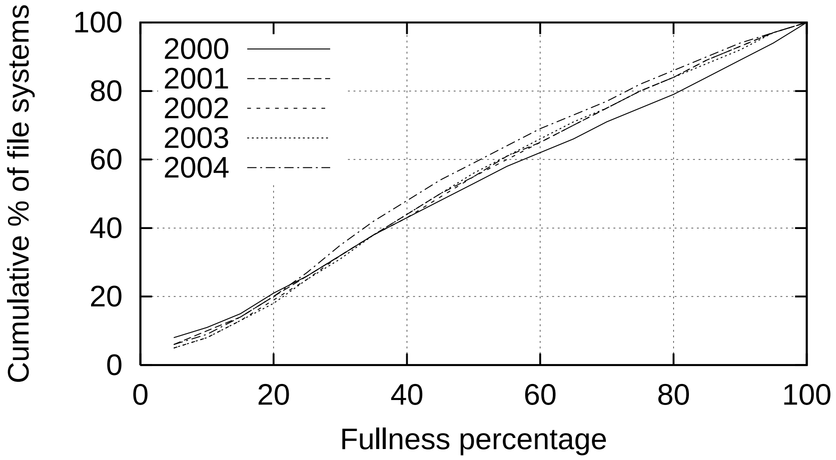


Figure 2.29 CDFs of file systems by fullness

be aware that some machines will have drastically more capacity than others. File system designs, which must last many years, must accommodate even more dramatic capacity differentials.

2.4.2 Changes in usage

This subsection describes our findings regarding how individual file systems change in fullness over time. For this part of our work, we examined the 6536 snapshot pairs that correspond to the same file system in two consecutive years. We also examined the 1320 snapshot pairs that correspond to the same file system two years apart. We find that 80% of file systems become fuller over a one-year period, and the mean increase in fullness is 14 percentage points. This increase is predominantly due to creation of new files, partly offset by deletion of old files, rather than due to extant files changing size.

When comparing two matching snapshots in different years, we must establish whether two files in successive snapshots of the same file system are the same file. We do not have access to files' inode numbers, because collecting them would have lengthened our scan times to an unacceptable degree. We thus instead use the following proxy for file

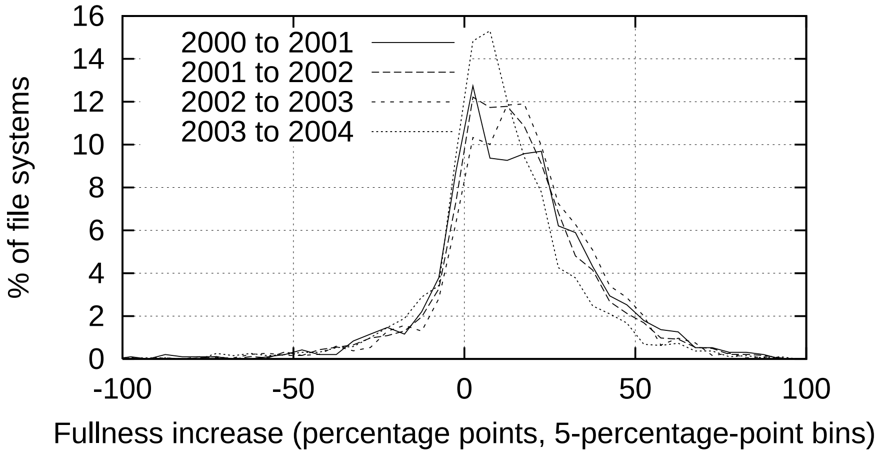


Figure 2.30 Histograms of file systems by 1-year fullness increase

sameness: If the files have the same full pathname, they are considered the same, otherwise they are not. This is a conservative approach: It will judge a file to be two distinct files if it or any ancestor directory has been renamed.

Figures 2.30 and 2.31 plot histograms and CDFs, respectively, of file systems by percentage-point increase in fullness from one year to the next. We define this term by example: If a file system was 50% full in 2000 and 60% full in 2001, it exhibited a 10 percentage-point increase in fullness. The distribution is substantially the same for all four pairs of consecutive years. Figure 2.31 shows that 80% of file systems exhibit an increase in fullness and fewer than 20% exhibit a decrease. The mean increase from one year to the next is 14 percentage points.

We also examined the increase in fullness over two years. We found the mean increase to be 22 percentage points. This is less than twice the consecutive-year increase, indicating that as file systems age, they increase their fullness at a slower rate. Because we have so few file systems with snapshots in four consecutive years, we did not explore increases over three or more years.

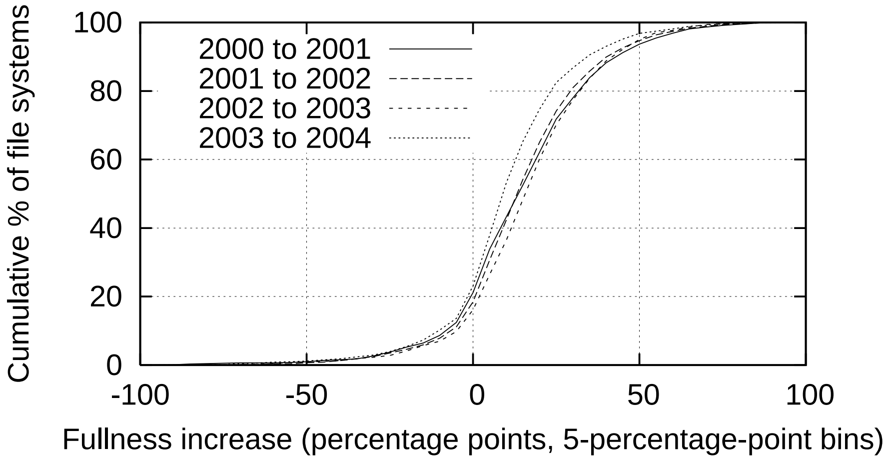


Figure 2.31 CDFs of file systems by 1-year fullness increase

Since file systems that persist for a year tend to increase their fullness by about 14 points, but the mean file-system fullness has dropped from 49% to 45% over our sample period, it seems that the steadily increasing fullness of individual file systems is offset by the replacement of old file systems with newer, emptier ones.

Analyzing the factors that contribute to the 14-point mean year-to-year increase in fullness revealed the following breakdown: Fullness increases by 28 percentage points due to files that are present in the later snapshot but not in the earlier one, meaning that they were created during the intervening year. Fullness decreases by 15 percentage points due to files that are present in the earlier snapshot but not in the later one, meaning that they were deleted during the intervening year. Fullness also increases by 1 percentage point due to growth in the size of files that are present in both snapshots. An insignificant fraction of this increase is attributable to changes in system paging files, internal fragmentation, or metadata storage.

We examined the size distributions of files that were created and of files that were deleted, to see if they differed from the overall file-size distribution. We found that they do not differ appreciably. We had hypothesized that users tend to delete large files to make room for new content, but the evidence does not support this hypothesis.

Since deleted files and created files have similar size distributions, file system designers need not expect the fraction of files of different sizes to change as a file system ages. Thus, if they find it useful to assign different parts of the disk to files of different sizes, they can anticipate the allocation of sizes to disk areas to not need radical change as time passes.

Many peer-to-peer systems use free space on computers to store shared data, so the amount of used space is of great importance. With an understanding of how this free space decreases as a file system ages, a peer-to-peer system can proactively plan how much it will need to offload shared data from each file system to make room for additional local content. Also, since a common reason for upgrading a computer is because its disk space becomes exhausted, a peer-to-peer system can use a prediction of when a file system will become full as a coarse approximation to when that file system will become unavailable.

2.5 Related Work

This research extends earlier work in measuring and modeling file-system metadata on Windows workstations. In 1998, Douceur and Bolosky collected snapshots of over ten thousand file systems on the desktop computers at Microsoft [41]. The focus of the earlier study was on variations among file systems within the sample, all of which were captured at the same time. By contrast, the focus of the present study is on longitudinal analysis, meaning how file systems have changed over time.

Prior to that previous study, there were no studies of static file-system metadata on Windows systems, but there were several such studies in other operating-system environments. These include Satyanarayanan's study of a Digital PDP-10 at CMU in 1981 [120], Mullender and Tanenbaum's study of a Unix system at Vrije Universiteit in 1984 [94], Irlam's study of 1050 Unix file systems in 1993 [67], and Sienknecht et al.'s study of 267 file systems in 46 HP-UX systems at Hewlett-Packard in 1994 [128]. All of these studies involved snapshots taken at a single time, like our study in 1998. There have also been longitudinal studies of file-system metadata, but for significantly shorter times than ours: Bennett *et al.*studied three file servers at the University of Western Ontario over a period of one day in 1991 [22], and Smith and Seltzer studied 48 file systems on four file servers at Harvard over a period of ten months in 1994 [131].

We are aware of only one additional collection of static file-system metadata since the previous study. In 2001, Evans and Kuenning captured snapshots from 22 machines running various operating systems at Harvey Mudd College and Marine Biological Laboratories [46]. Their data collection and analysis focused mainly, but not exclusively, on media files. Their findings show that different types of files exhibit significantly different size distributions, which our results support.

Many studies have examined dynamic file-system traces rather than static file system snapshots. These studies are complementary to ours, describing things we cannot analyze such as the rate at which bytes are read and written in a file system. A few examples of such studies are Ousterhout *et al.*'s analysis of the BSD file system [101], Gribble *et al.*'s analysis of self-similarity in the dynamic behavior of various file systems [58], Vogels's analysis of Windows NT [153], and Roselli *et al.*'s analysis of HP-UX and Windows NT [116].

In addition to file-system measurement research, there has been much work in modeling file-system characteristics, most notably related to the distribution of file sizes. Examples of work in this area include that of Satyanarayanan [120], Barford and Crovella [18], Downey [42], and Mitzenmacher [91].

In 2001, Evans and Kuenning broke down measured file-size distributions according to file type, and they modeled the sizes using log-lambda distributions [46]. They found that video and audio files can significantly perturb the file-size distribution and prevent simple size models from applying. We did not find this to be true for file sizes in our sample population. However, we did find video, database, and blob files responsible for a second peak in the distribution of bytes by containing file size.

In the previous study by Douceur and Bolosky, directory depth was modeled with a Poisson distribution [41], but we have herein proposed a generative model in which the attractiveness of an extant directory d as a location for a new subdirectory is proportional to $c(d) + 2$, where $c(d)$ is the count of directory d's extant subdirectories. This is strikingly similar to the rule for generating plane-oriented recursive trees, wherein the probability is proportional to $c(d) + 1$ [80]. The generative model has the added advantage of being easy to implement computationally.

2.6 Discussion

Over a span of five years, we collected metadata snapshots from more than 63,000 distinct Windows file systems in a commercial environment, through voluntary participation of the systems' users. These systems contain 4 billion files totaling 700 TB of file data. For more than 10% of these file systems, we obtained snapshots in multiple years. Since these snapshots from multiple years were from the same general population, it enabled us to directly observe how these file systems have changed over time. Our measurements reveal several interesting properties of file systems and offer useful lessons.

One interesting discovery is the emergence of a second mode in the GB range in the distribution of bytes by containing file size. It makes us wonder if at some future time a third mode will arise. The increasingly large fraction of content in large files suggests that variable block sizes, as supported by ZFS [24] and NTFS [136], are becoming increasingly important. Since a few large files, mainly video, database, and blob files, are contributing to an increasing fraction of file-system usage, these file extensions are ideal candidates for larger block sizes.

Although large files account for a large fraction of space, most files are 4 KB or smaller. Thus, it is useful to co-locate several small files in a single block, as ReiserFS [110] does, and to co-locate small file content with file metadata, as NTFS does. Our finding that most directories have few entries suggests yet another possibility: Co-locate small file content with the file's parent directory. An even more extreme solution is suggested by the fact that in 2004, the average file system had only 52 MB in files 4 KB or smaller. Since this number is becoming small relative to main memory sizes, it may soon be practical to avoid cache misses entirely for small files by prefetching them all at boot time and pinning them in the cache.

Another noteworthy discovery is that the fraction of files locally modified decreases with time, an effect significant enough to be observable in only a five-year sample. It would appear that users' ability to generate increasing amounts of content is outstripped by the phenomenal growth in their disks. If individuals copying content from each other becomes increasingly common, then applications like peer-to-peer backup will have increasing amounts of inter-machine content similarity to leverage to obviate copying.

We were surprised to find a strong negative correlation between namespace depth and file size. Such a strong and temporally-invariant correlation, in combination with the well-known correlation between file extension and file size, can help us make predictions of file size at creation time. This may be useful, e.g., to decide how many blocks to initially allocate to a file.

We also discovered that a simple generative model can account for both the distributions of directory depth and the count of subdirectories per directory. The model we developed posits that new subdirectories are created inside an existing directory in offset proportion to the count of subdirectories already in that directory. This behavior is easy to simulate, and it produces directory-depth and directory-size distributions that closely match our observations.

Finally, it is remarkable that file system fullness over the course of five years has changed little despite the vast increase in file system capacity over that same period. It seems clear that users scale their capacity needs to their available capacity. The lesson for storage manufacturers is to keep focusing effort on increasing capacity, because customers will continue to place great value on capacity for the foreseeable future.

2.7 Conclusion

Developers of file systems and related software utilities frequently rely on information about file system usage in real settings. Such information is invaluable for designing and evaluating new and existing systems. In this chapter we presented our longitudinal study of file-system metadata wherein we find significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities. Wherever applicable, we gave examples of consequent lessons for designers of file system software.

We have made our traces available to the community via the Storage Networking Industry Association's IOTTA repository. To obtain them, visit the URL `http://iotta.snia.org/traces/tracesStaticSnapshot/`.

Chapter 3

Generating Realistic *Impressions* for File-System Benchmarking

One of the most important challenges in file-system benchmarking apart from creating representative benchmark workloads is to recreate the file-system *state* such that it is representative of the target usage scenario. Several factors contribute to file-system state, important amongst them are the *in-memory* state (contents of the buffer cache), the *on-disk* state (disk layout and fragmentation) and the characteristics of the *file-system image* (files and directories belonging to the namespace and file contents).

One well understood contributor to state is the *in-memory* state of the file system. Previous work has shown that the contents of the cache can have significant impact on the performance results [38]. Therefore, system initialization during benchmarking typically consists of a cache "warm-up" phase wherein the workload is run for some time prior to the actual measurement phase. Another important factor is the *on-disk* state of the file system, or the degree of *fragmentation*; it is a measure of how the disk blocks belonging to the file system are laid out on disk. Previous work has shown that fragmentation can adversely affect performance of a file system [132]. Thus, prior to benchmarking, a file system should undergo *aging* by replaying a workload similar to that experienced by a real file system over a period of time [132].

Surprisingly, one key contributor to file-system state has been largely ignored – the characteristics of the *file-system image*. The properties of file-system metadata and the

Paper	Description	Used to measure
HAC [55]	File system with 17000 files totaling 150 MB	Time and space needed to create a Glimpse index
IRON [106]	None provided	Checksum and metadata replication overhead; parity block overhead for user files
LBFS [95]	10702 files from /usr/local, total size 354 MB	Performance of LBFS chunking algorithm
LISFS [102]	633 MP3 files, 860 program files, 11502 man pages	Disk space overhead; performance of search-like activities: UNIX find and LISFS lookup
PAST [117]	2 million files, mean size 86 KB, median 4 KB, largest file size 2.7 GB, smallest 0 Bytes, total size 166.6 GB	File insertion, global storage utilization in a P2P system
Pastiche [36]	File system with 1641 files, 109 dirs, 13.4 MB total size	Performance of backup and restore utilities
Pergamum [139]	Randomly generated files of “several” megabytes	Data transfer performance
Samsara [37]	File system with 1676 files and 13 MB total size	Data transfer and querying performance, load during querying
Segank [134]	5-deep directory tree, 5 subdirs and 10 8 KB files per directory	Performance of Segank: volume update, creation of read-only snapshot, read from new snapshot
SFS read-only [48]	1000 files distributed evenly across 10 directories and contain random data	Single client/single server read performance
TFS [33]	Files taken from /usr to get “realistic” mix of file sizes	Performance with varying contribution of space from local file systems
WAFL backup [66]	188 GB and 129 GB volumes taken from the Engineering department	Performance of physical and logical backup, and recovery strategies
yFS [161]	Avg. file size 16 KB, avg. number of files per directory 64, random file names	Performance under various benchmarks (file creation, deletion)

Table 3.1 **Choice of file system parameters in prior research.**

actual content within the files are key contributors to file-system state, and can have a significant impact on the performance of a system. Properties of file-system metadata includes information on how directories are organized in the file-system namespace, how files are organized into directories, and the distributions for various file attributes such as size, depth, and extension type. Consider a simple example: the time taken for a `find` operation to traverse a file system while searching for a file name depends on a number of attributes of the file-system image, including the depth of the file-system tree and the total number of files. Similarly, the time taken for a `grep` operation to search for a keyword also depends on the type of files (*i.e.*, binary vs. others) and the file content.

File-system benchmarking frequently requires this sort of information on file systems, much of which is available in the form of empirical studies of file-system contents [8, 41, 67, 94, 120, 128], such as the one presented in Chapter 2 that was conducted by us. Such studies focus on measuring and modeling different aspects of file-system metadata by collecting snapshots of file-system images from real machines. The studies range from a few machines to tens of thousands of machines across different operating systems and usage environments. Collecting and analyzing this data provides useful information on how file systems are used in real operating conditions.

In spite of the wealth of information available in file-system studies, system designers and evaluators continue to rely on *ad hoc* assumptions and often inaccurate rules of thumb. Table 3.1 presents evidence to confirm this hypothesis; it contains a (partial) list of publications from top-tier systems conferences in the last ten years that required a test file-system image for evaluation. We present both the description of the file-system image provided in the paper and the intended goal of the evaluation.

In the table, there are several examples where a new file system or application design is evaluated on the evaluator's personal file system without describing its properties in sufficient detail for it to be reproduced [33, 66, 106]. In others, the description is limited to coarse-grained measures such as the total file-system size and the number of files, even though other file-system attributes (*e.g.*, tree depth) are relevant to measuring performance or storage space overheads [36, 37, 55, 95]. File systems are also sometimes generated with parameters chosen randomly [139, 161], or chosen without explanation of the significance of the values [48, 102, 134]. Occasionally, the parameters are specified in greater detail [117], but not enough to recreate the original file system.

The important lesson to be learnt here is that there is no standard technique to systematically include information on file-system images for experimentation. For this reason, we find that more often than not, the choices made are arbitrary, suited for ease-of-use more than accuracy and completeness. Furthermore, the lack of standardization and reproducibility of these choices makes it near-impossible to compare results with other systems.

To address these problems and improve one important aspect of file system benchmarking, we develop *Impressions*, a framework to generate representative and statistically accurate file-system images. Impressions gives the user flexibility to specify one or more parameters from a detailed list of file system parameters (file-system size, number of files, distribution of file sizes, etc.). Impressions incorporates statistical techniques (automatic curve-fitting, resolving multiple constraints, interpolation and extrapolation, etc.) and uses statistical tests for goodness-of-fit to ensure the accuracy of the image.

We believe Impressions will be of great use to system designers, evaluators, and users alike. A casual user looking to create a representative file-system image without worrying about carefully selecting parameters can simply run Impressions with its default settings; Impressions will use pre-specified distributions from file-system studies to create a representative image. A more sophisticated user has the power to individually control the knobs for a comprehensive set of file-system parameters; Impressions will carefully work out the statistical details to produce a consistent and accurate image. In both cases, Impressions ensures complete reproducibility of the image, by reporting the used distributions, parameter values, and seeds for random number generators.

In this chapter we present the design, implementation and evaluation of the Impressions framework (§3.2). Impressions is built with the following design goals:

- *Accuracy:* in generating various statistical constructs to ensure a high degree of statistical rigor.
- *Flexibility:* in allowing users to specify a number of file-system distributions and constraints on parameter values, or in choosing default values.
- *Representativeness:* by incorporating known distributions from file-system studies.
- *Ease of use:* by providing a simple, yet powerful, command-line interface.

Using desktop search as a case study, we then demonstrate the usefulness and ease of use of Impressions in quantifying application performance, and in finding application policies and bugs (§3.3).

3.1 Extended Motivation

We begin this section by asking a basic question: does file-system structure really matter? We then describe the goals for generating realistic file-system images and discuss existing approaches to do so.

3.1.1 Does File-System Structure Matter?

Structure and organization of file-system metadata matters for workload performance. Let us take a look at the simple example of a frequently used UNIX utility: `find`. Figure 3.1 shows the relative time taken to run "`find /`" searching for a file name on a test file system as we vary some parameters of file-system state.

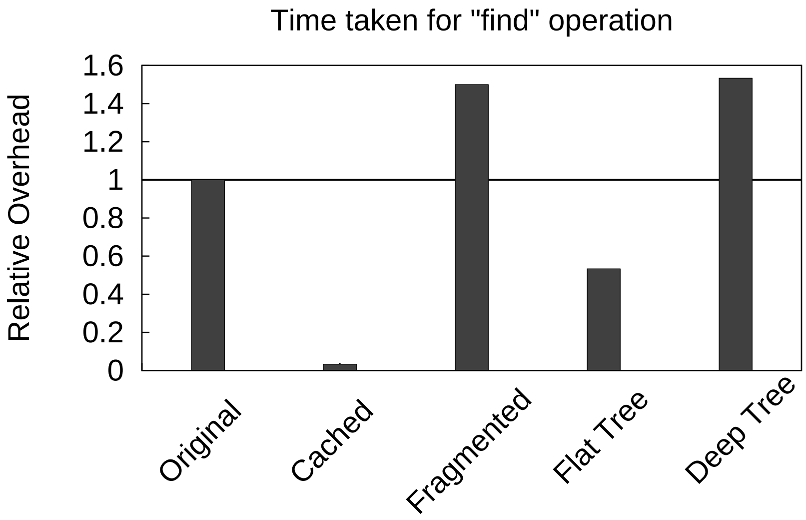


Figure 3.1 **Impact of directory tree structure.** *Shows impact of tree depth on time taken by* find*. The file systems are created by Impressions using default distributions (Table 3.2). To exclude effects of the on-disk layout, we ensure a perfect disk layout (layout score* 1.0*) for all cases except the one with fragmentation (layout score* 0.95*). The* flat tree *contains all* 100 *directories at depth* 1*; the* deep tree *has directories successively nested to create a tree of depth* 100.

The first bar represents the time taken for the run on the original test file system. Subsequent bars are normalized to this time and show performance for a run with the file-system contents in buffer cache, a fragmented version of the same file system, a file system created by flattening the original directory tree, and finally one by deepening the original directory tree. The graph echoes our understanding of caching and fragmentation, and brings out one aspect that is often overlooked: structure really matters. From this graph we can see that even for a simple workload, the impact of tree depth on performance can be as large as that with fragmentation, and varying tree depths can have significant performance variations (300% between the flat and deep trees in this example).

Assumptions about file-system structure have often trickled into file system design, but no means exist to incorporate the effects of realistic file-system images in a systematic fashion. As a community, we well understand that caching matters, and have begun to pay attention to fragmentation, but when it comes to file-system structure, our approach is surprisingly *laissez faire*.

3.1.2 Goals for Generating FS Images

We believe that the file-system image used for an evaluation should be *realistic* with respect to the workload; the image should contain a sufficient degree of *detail* to realistically exercise the workload under consideration. An increasing degree of detail will likely require more effort and slow down the process. Thus it is useful to know the degree sufficient for a given evaluation. For example, if the performance of an application simply depends on the size of files in the file system, the chosen file-system image should reflect that. On the other hand, if the performance is also sensitive to the fraction of binary files amongst all files (*e.g.*, to evaluate desktop search indexing), then the file-system image also needs to contain realistic distributions of file extensions.

We walk through some examples that illustrate the different degrees of detail needed in file-system images.

- At one extreme, a system could be completely oblivious to both metadata and content. An example of such a system is a mirroring scheme (RAID-1 [103]) underneath a file system, or a backup utility taking whole-disk backups. The performance of such schemes depends solely on the block traffic.

Alternately, systems could depend on the attributes of the file-system image with different degrees of detail:

- The performance of a system can depend on the amount of file data (number of files and directories, or the size of files and directories, or both) in any given file system (*e.g.*, a backup utility taking whole file-system snapshots).
- Systems can depend on the structure of the file system namespace and how files are organized in it (*e.g.*, a version control system for a source-code repository).
- Finally, many systems also depend on the actual data stored within the files (*e.g.*, a desktop search engine for a file system, or a spell-checker).

Impressions is designed with this goal of flexibility from the outset. The user is given complete control of a number of file-system parameters, and is provided with an easy to use interface. Transparently, Impressions seamlessly ensures accuracy and representativeness.

3.1.3 Existing Approaches

One alternate approach to generating realistic file-system images is to randomly select a set of actual images from a corpus, an approach popular in other fields of computer science such as Information Retrieval, Machine Learning and Natural Language Processing [97]. In the case of file systems the corpus would consist of a set of known file-system images. This approach arguably has several limitations which make it difficult and unsuitable for file systems research. First, there are too many parameters required to accurately describe a file-system image that need to be captured in a corpus. Second, without precise control in varying these parameters according to experimental needs, the evaluation can be blind to the actual performance dependencies. Finally, the cost of maintaining and sharing any realistic corpus of file-system images would be prohibitive. The size of the corpus itself would severely restrict its usefulness especially as file systems continue to grow larger.

Unfortunately, these limitations have not deterred researchers from using their personal file systems as a (trivial) substitute for a file-system corpus.

3.2 The Impressions Framework

In this section we describe the design, implementation and evaluation of Impressions: a framework for generating file-system images with realistic and statistically accurate metadata and content. Impressions is flexible enough to create file-system images with varying configurations, guaranteeing the accuracy of images by incorporating a number of statistical tests and techniques.

We first present a summary of the different modes of operation of Impressions, and then describe the individual statistical constructs in greater detail. Wherever applicable, we evaluate their accuracy and performance.

3.2.1 Modes of Operation

A system evaluator can use Impressions in different modes of operation, with varying degree of user input. Sometimes, an evaluator just wants to create a representative file-system image without worrying about the need to carefully select parameters. Hence, in the *automated* mode, Impressions is capable of generating a file-system image with minimal input required from the user (*e.g.*, the size of the desired file-system image), relying on default settings of known empirical distributions to generate representative file-system images. We refer to these distributions as *original* distributions.

At other times, users want more control over the images, for example, to analyze the sensitivity of performance to a given file-system parameter, or to describe a completely different file-system usage scenario. Hence, Impressions supports a *user-specified* mode, where a more sophisticated user has the power to individually control the knobs for a comprehensive set of file-system parameters; we refer to these as user-specified distributions. Impressions carefully works out the statistical details to produce a consistent and accurate image. In both the cases, Impressions ensures complete reproducibility of the file-system

Parameter	Default Model & Parameters
Directory count w/ depth	Generative model
Directory size (subdirs)	Generative model
File size by count	Lognormal-body
	(α_1=0.99994, μ=9.48, σ=2.46)
	Pareto-tail (k=0.91,$\mathcal{X}_m$=512MB)
File size by containing bytes	Mixture-of-lognormals
	(α_1=0.76, μ_1=14.83, σ_1=2.35
	α_2=0.24, μ_2=20.93, σ_2=1.48)
Extension popularity	Percentile values
File count w/ depth	Poisson (λ=6.49)
Bytes with depth	Mean file size values
Directory size (files)	Inverse-polynomial
	(degree=2, offset=2.36)
File count w/ depth (w/ special directories)	Conditional probabilities (biases for special dirs)
Degree of Fragmentation	Layout score (1.0)
	or Pre-specified workload

Table 3.2 **Parameters and default values in Impressions.** *List of distributions and their parameter values used in the Default mode.*

image by reporting the used distributions, their parameter values, and seeds for random number generators.

Impressions can use any dataset or set of parameterized curves for the *original* distributions, leveraging a large body of research on analyzing file-system properties [8, 41, 67, 94, 120, 128]. For illustration, in this dissertation we use the findings presented in Chapter 2 from our study on file system metadata, and the corresponding snapshot dataset that was made publicly available. To briefly summarize the study presented in the previous chapter, the snapshots of file-system metadata were collected over a five-year period representing over $60,000$ Windows PC file systems in a large corporation. These snapshots were then used to study distributions and temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity, and degree of file modification. The study also proposed a generative model explaining the creation of file-system namespaces.

Impressions provides a comprehensive set of individually controllable file system parameters. Table 3.2 lists these parameters along with their default selections. For example, a user may specify the size of the file-system image, the number of files in the file system, and the distribution of file sizes, while selecting default settings for all other distributions. In this case, Impressions will ensure that the resulting file-system image adheres to the default distributions while maintaining the user-specified invariants.

The default values listed in this table are derived from the corresponding data presented in Chapter 2 for the year 2004. The models for directory count with depth and directory size with subdirectories correspond to the generative model previously discussed in Section §2.3.5; the entries for file size by count and by containing bytes refer to Figures 2.3 and 2.5; extension popularity is based on the Figures 2.9 and 2.10; the values for file count with depth and bytes with depth are derived from the Figures 2.19 and 2.21; directory size in the number of files is from the Figure 2.14; the biases for file counts with depth including special directories corresponds to data from the Figure 2.17; the degree of fragmentation is specified by the user arbitrarily, or according to pre-specified workloads that execute a set number of iterations of a known workload.

3.2.2 Basic Techniques

The goal of Impressions is to generate realistic file-system images, giving the user complete flexibility and control to decide the extent of accuracy and detail. To achieve this, Impressions relies on a number of statistical techniques.

In the simplest case, Impressions needs to create statistically accurate file-system images with default distributions. Hence, a basic functionality required by Impressions is to convert the parameterized distributions into real sample values used to create an instance of a file-system image. Impressions uses random sampling to take a number of independent observations from the respective probability distributions. Wherever applicable, such parameterized distributions provide a highly compact and easy-to-reproduce representation of observed distributions. For cases where standard probability distributions are infeasible, a Monte Carlo method is used.

A user may want to use file system datasets other than the default choice. To enable this, Impressions provides automatic curve-fitting of empirical data.

Impressions also provides the user with the flexibility to specify distributions and constraints on parameter values. One challenge thus is to ensure that multiple constraints specified by the user are resolved consistently. This requires statistical techniques to ensure that the generated file-system images are accurate with respect to both the user-specified constraints and the default distributions.

In addition, the user may want to explore values of file system parameters, not captured in any dataset. For this purpose, Impressions provides support for interpolation and extrapolation of new curves from existing datasets.

Finally, to ensure the accuracy of the generated image, Impressions contains a number of built-in statistical tests, for goodness-of-fit (*e.g.*, Kolmogorov-Smirnov, Chi-Square, and Anderson-Darling), and to estimate error (*e.g.*, Confidence Intervals, MDCC, and Standard Error). Where applicable, these tests ensure that all curve-fit approximations and internal statistical transformations adhere to the highest degree of statistical rigor desired.

3.2.3 Creating Valid Metadata

The simplest use of Impressions is to generate file-system images with realistic metadata. This process is performed in two phases: first, the skeletal file-system namespace is created; and second, the namespace is populated with files conforming to a number of file and directory distributions.

3.2.3.1 Creating File-System Namespace

The first phase in creating a file system is to create the namespace structure or the *directory tree*. We assume that the user specifies the size of the file-system image. The count of files and directories is then selected based on the file system size (if not specified by the user). Depending on the degree of detail desired by the user, each file or directory attribute is selected step by step until all attributes have been assigned values. We now describe this process assuming the highest degree of detail.

To create directory trees, Impressions uses the generative model proposed by Agrawal *et al.* [8] to perform a Monte Carlo simulation. According to this model, new directories are added to a file system one at a time, and the probability of choosing each extant directory as a parent is proportional to $\mathcal{C}(d) + 2$, where $\mathcal{C}(d)$ is the count of extant subdirectories of directory d. The model explains the creation of the file system namespace, accounting both for the size and count of directories by depth, and the size of parent directories. The input to this model is the total number of directories in the file system. Directory names are generated using a simple iterative counter.

To ensure the accuracy of generated images, we compare the generated distributions (*i.e.*, created using the parameters listed in Table 3.2), with the desired distributions (*i.e.*, ones obtained from the dataset discussed previously in Chapter 2 corresponding to year 2004). Figures 3.2 and 3.3 show in detail the accuracy for each step in the namespace and file creation process. For almost all the graphs, the y-axis represents the percentage of files, directories, or bytes belonging to the categories or bins shown on the x-axis, as the case may be.

Figures 3.2(a) and 3.2(b) show the distribution of directories by depth, and directories by subdirectory count, respectively. The y-axis in this case is the percentage of directories at each level of depth in the namespace, shown on the x-axis. The two curves representing the generated and the desired distributions match quite well indicating good accuracy.

3.2.3.2 Creating Files

The next phase is to populate the directory tree with files. Impressions spends most of the total runtime and effort during this phase, as the bulk of its statistical machinery is exercised in creating files. Each file has a number of attributes such as its size, depth in the directory tree, parent directory, and file extension. Similarly, the choice of the parent directory is governed by directory attributes such as the count of contained subdirectories, the count of contained files, and the depth of the parent directory. Analytical approximations for file system distributions proposed previously [41] guided our own models.

First, for each file, the size of the file is sampled from a hybrid distribution describing file sizes. The body of this hybrid curve is approximated by a lognormal distribution, with a Pareto tail distribution (k=0.91, $\mathcal{X}_m$=512MB) accounting for the heavy tail of files with size greater than 512 MB. The exact parameter values used for these distributions are listed in Table 3.2. These parameters were obtained by fitting the respective curves to file sizes obtained from the file-system dataset previously discussed in Chapter 2. Figure 3.2(c) shows the accuracy of generating the distribution of files by size. We initially used a simpler model for file sizes represented solely by a lognormal distribution. While the results were acceptable for files by size (Figure 3.2(c)), the simpler model failed to account for the distribution of bytes by containing file size; coming up with a model to accurately capture the bimodal distribution of bytes proved harder than we had anticipated. Figure 3.2(d) shows the accuracy of the hybrid model in Impressions in generating the distribution of bytes. The pronounced double mode observed in the distribution of bytes is a result of the

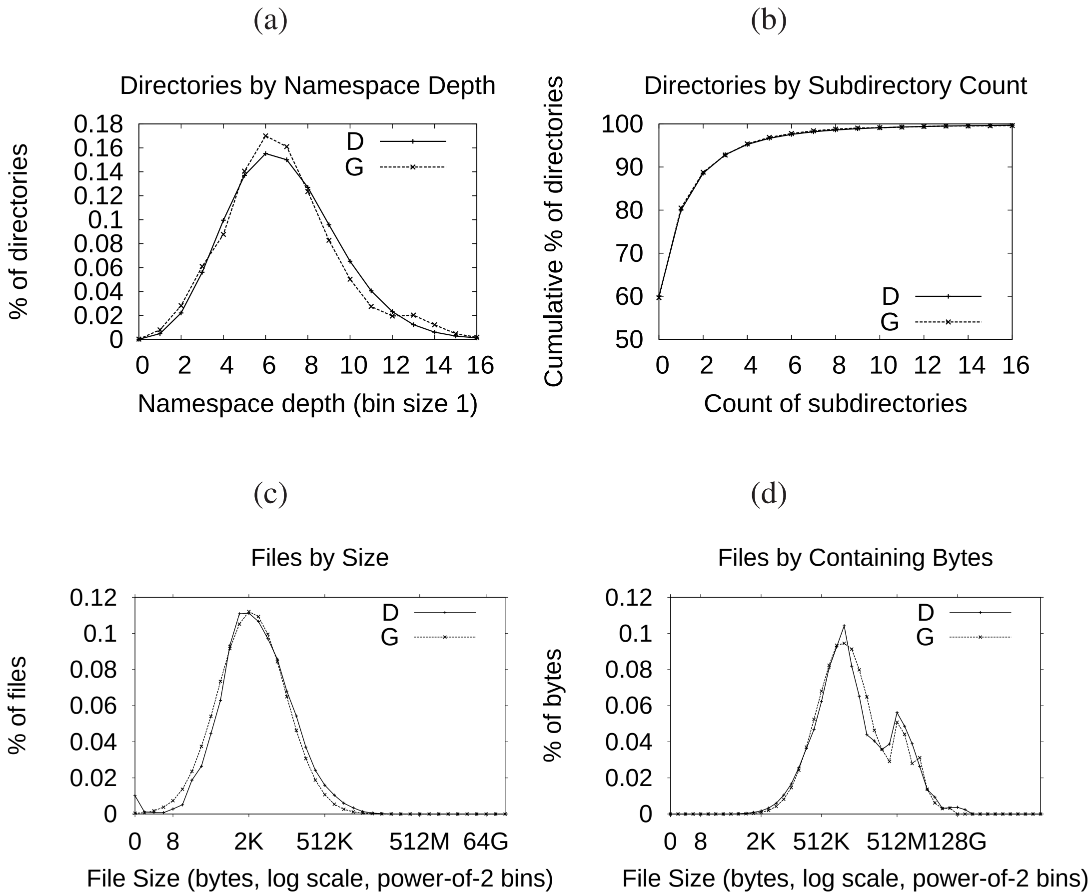


Figure 3.2 **Accuracy of Impressions in recreating file system properties.** *Shows the accuracy of the entire set of file system distributions modeled by Impressions. D: the desired distribution; G: the generated distribution. Impressions is quite accurate in creating realistic file system state for all parameters of interest shown here. We include a special abscissa for the zero value on graphs having a logarithmic scale.*

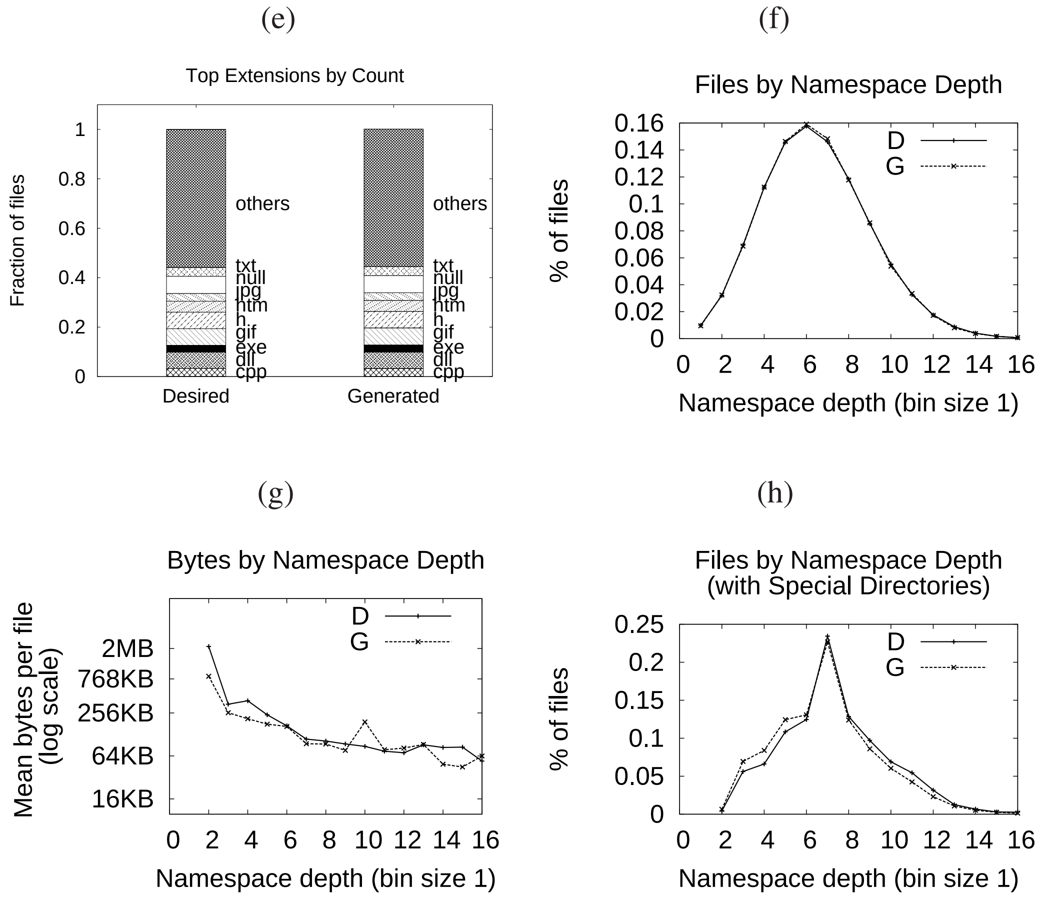


Figure 3.3 **Accuracy of Impressions in recreating file system properties.** *Shows the accuracy of the entire set of file system distributions modeled by Impressions. D: the desired distribution; G: the generated distribution. Impressions is quite accurate in creating realistic file system state for all parameters of interest shown here. We include a special abscissa for the zero value on graphs having a logarithmic scale.*

presence of a few large files; an important detail that is otherwise missed if the heavy-tail of file sizes is not accurately accounted for.

Once the file size is selected, we assign the file name and extension. Impressions keeps a list of percentile values for popular file extensions (*i.e.*, top 20 extensions by count, and by bytes). These extensions together account for roughly 50% of files and bytes in a file system ensuring adequate coverage for the important extensions. The remainder of files are given randomly generated three-character extensions. Currently filenames are generated by a simple numeric counter incremented on each file creation. Figure 3.3(e) shows the accuracy of Impressions in creating files with popular extensions by count.

Next, we assign file depth d, which requires satisfying two criteria: the distribution of files with depth, and the distribution of bytes with depth. The former is modeled by a Poisson distribution, and the latter is represented by the mean file sizes at a given depth. Impressions uses a multiplicative model combining the two criteria, to produce appropriate file depths. Figures 3.3(f) and 3.3(g) show the accuracy in generating the distribution of files by depth, and the distribution of bytes by depth, respectively.

The final step is to select a parent directory for the file, located at depth $d-1$, according to the distribution of directories with file count, modeled using an inverse-polynomial of degree 2. As an added feature, Impressions supports the notion of "Special" directories containing a disproportionate number of files or bytes (*e.g.*, "Program Files" folder in the Windows environment). If required, during the selection of the parent directory, a selection bias is given to these special directories. Figure 3.3(h) shows the accuracy in supporting special directories with an example of a *typical* Windows file system having files in the web cache at depth 7, in `Windows` and `Program Files` folders at depth 2, and `System` files at depth 3.

Parameter	MDCC
Directory count with depth	0.03
Directory size (subdirectories)	0.004
File size by count	0.04
File size by containing bytes	0.02
Extension popularity	0.03
File count with depth	0.05
Bytes with depth	0.12 MB*
File count w/ depth w/ special dirs	0.06

Table 3.3 **Statistical accuracy of generated images.** *Shows average accuracy of generated file-system images in terms of the MDCC (Maximum Displacement of the Cumulative Curves) representing the maximum difference between cumulative curves of generated and desired distributions. Averages are shown for* 20 *trials. (*) For bytes with depth, MDCC is not an appropriate metric, we instead report the average difference in mean bytes per file (MB). The numbers correspond to the set of graphs shown in Figure 3.3 and reflect fairly accurate images.*

Table 3.3 shows the average difference between the generated and desired images from Figure 3.3 for 20 trials. The difference is measured in terms of the MDCC (Maximum Displacement of the Cumulative Curves). For instance, an MDCC value of 0.03 for directories with depth, implies a *maximum* difference of 3% on an average, between the desired and the generated cumulative distributions. Overall, we find that the models created and used by Impressions for representing various file-system parameters produce fairly accurate distributions in all the above cases. While we have demonstrated the accuracy of Impressions for the Windows dataset, there is no fundamental restriction limiting it to this dataset. We believe that with little effort, the same level of accuracy can be achieved for any other dataset.

3.2.4 Resolving Arbitrary Constraints

One of the primary requirements for Impressions is to allow flexibility in specifying file system parameters without compromising accuracy. This means that users are allowed

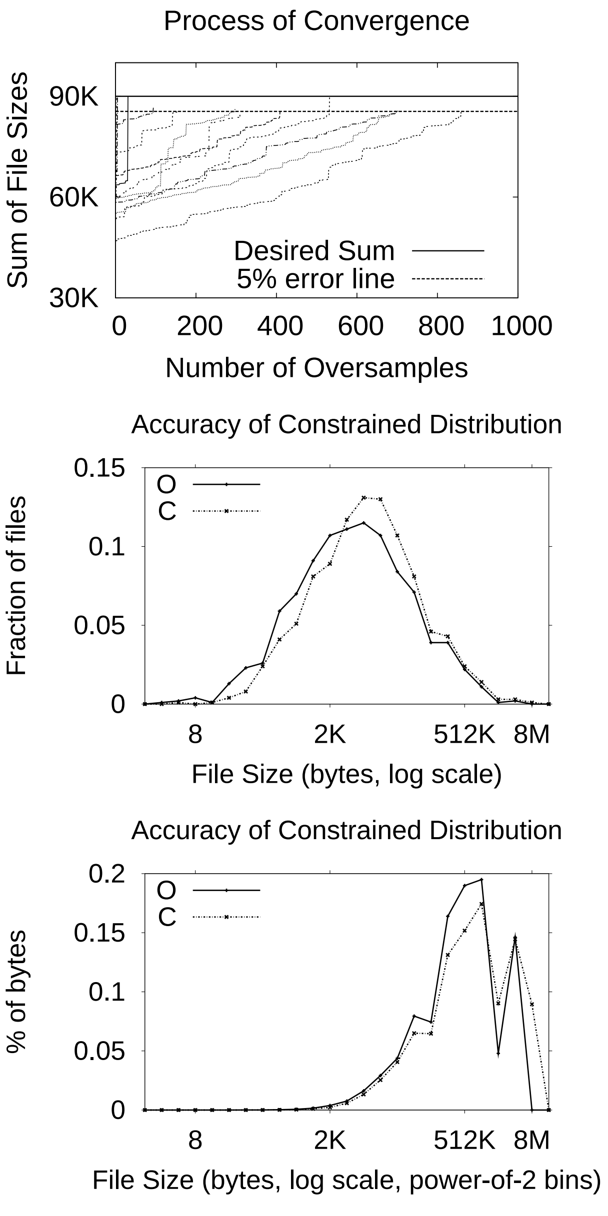


Figure 3.4 **Resolving Multiple Constraints.** *(a) Shows the process of convergence of a set of 1000 file sizes to the desired file system size of 90000 bytes. Each line represents an individual trial. A successful trial is one that converges to the 5% error line in less than 1000 oversamples. (b) Shows the difference between the original distribution of files by size, and the constrained distribution after resolution of multiple constraints in (a). O: Original; C: Constrained. (c) Same as (b), but for distribution of files by bytes instead.*

Num. files $\mathcal{N}$	File sizes sum $\mathcal{S}$ (bytes)	Avg. β Initial	Avg. β Final	Avg. α	Avg. D Count	Avg. D Bytes	Success
1000	30000	21.55%	2.04%	5.74%	0.043	0.050	100%
1000	60000	20.01%	3.11%	4.89%	0.032	0.033	100%
1000	90000	34.35%	4.00%	41.2%	0.067	0.084	90%

Table 3.4 **Summary of resolving multiple constraints.** *Shows average rate and accuracy of convergence after resolving multiple constraints for different values of desired file system size generated with a lognormal file size distribution* $\mathcal{D}_3$ *(*μ=8.16, σ=2.46*).* β*: % error between the desired and generated sum,* α*: % of oversamples required,* D *is the test statistic for the K-S test representing the maximum difference between generated and desired empirical cumulative distributions. Averages are for* 20 *trials. Success is the number of trials having final* $\beta \leq 5\%$*, and* D *passing the K-S test.*

to specify somewhat arbitrary constraints on these parameters, and it is the task of Impressions to resolve them. One example of such a set of constraints would be to specify a large number of files for a small file system, or vice versa, given a file size distribution. Impressions will try to come up with a sample of file sizes that best approximates the desired distribution, while still maintaining the invariants supplied by the user, namely the number of files in the file system and the sum of all file sizes being equal to the file system used space.

Multiple constraints can also be implicit (*i.e.*, arise even in the absence of user-specified distributions). Due to random sampling, different sample sets of the same distribution are not guaranteed to produce exactly the same result, and consequently, the sum of the elements can also differ across samples. Consider the previous example of file sizes again: the sum of all file sizes drawn from a given distribution need not add up to the desired file system size (total used space) each time. More formally, this example is represented by the following set of constraints:

$$
\begin{aligned}
\mathcal{N} &= \{Constant_1 \vee x : x \in \mathcal{D}_1(x)\} \\
\mathcal{S} &= \{Constant_2 \vee x : x \in \mathcal{D}_2(x)\} \\
\mathcal{F} = \{x : x \in \mathcal{D}_3(x; \mu, \sigma)\};\ \mid & \sum_{i=0}^{\mathcal{N}} \mathcal{F}_i - \mathcal{S} \mid \leq \beta * \mathcal{S}
\end{aligned}
\tag{3.1}
$$

where $\mathcal{N}$ is the number of files in the file system; $\mathcal{S}$ is the desired file system used space; $\mathcal{F}$ is the set of file sizes; and β is the maximum relative error allowed. The first two constraints specify that $\mathcal{N}$ and $\mathcal{S}$ can be user specified constants or sampled from their corresponding distributions $\mathcal{D}_1$ and $\mathcal{D}_2$. Similarly, $\mathcal{F}$ is sampled from the file size distribution $\mathcal{D}_3$. These attributes are further subject to the constraint that the sum of all file sizes differs from the desired file system size by no more than the allowed error tolerance, specified by the user. To solve this problem, we use the following two techniques:

- If the initial sample does not produce a result satisfying all the constraints, we *over-sample* additional values of $\mathcal{F}$ from $\mathcal{D}_3$, one at a time, until a solution is found, or the oversampling factor $\alpha/\mathcal{N}$ reaches λ (the maximum oversampling factor). α is the count of extra samples drawn from $\mathcal{D}_3$. Upon reaching λ without finding a solution, we discard the current sample set and start over.
- The number of elements in $\mathcal{F}$ during the oversampling stage is $\mathcal{N} + \alpha$. For every oversampling, we need to find if there exists $\mathcal{F}_{Sub}$, a subset of $\mathcal{F}$ with $\mathcal{N}$ elements, such that the sum of all elements of $\mathcal{F}_{Sub}$ (file sizes) differs from the desired file system size by no more than the allowed error. More formally stated, we find if:

$$\exists\, \mathcal{F}_{Sub} = \{\mathcal{X} : \mathcal{X} \subseteq \mathbb{P}(F),\ |\mathcal{X}| = \mathcal{N},\ |\mathcal{F}| = \mathcal{N} + \alpha,$$
$$|\sum_{i=0}^{\mathcal{N}} \mathcal{X}_i - \mathcal{S}\,| \leq \beta * \mathcal{S},\ \alpha \in \mathbb{N} \wedge \frac{\alpha}{\mathcal{N}} \leq \lambda\} \tag{3.2}$$

The problem of resolving multiple constraints as formulated above, is a variant of the more general "Subset Sum Problem" which is NP-complete [34]. Our solution is thus an approximation algorithm based on an existing $O(n \log n)$ solution [107] for the Subset Sum Problem.

The existing algorithm has two phases. The first phase randomly chooses a solution vector which is valid (the sum of elements is less than the desired sum), and maximal (adding any element not already in the solution vector will cause the sum to exceed the desired sum). The second phase performs *local improvement*: for each element in the solution, it searches for the largest element not in the current solution which, if replaced with the current element, would reduce the difference between the desired and current sums. The solution vector is updated if such an element is found, and the algorithm proceeds with the next element, until all elements are compared.

Our problem definition and the modified algorithm differ from the original in the following ways:

- First, in the original problem, there is no restriction on the number of elements in the solution subset $\mathcal{F}_{Sub}$. In our case, $\mathcal{F}_{Sub}$ can have exactly $\mathcal{N}$ elements. We modify the first phase of the algorithm to set the initial $\mathcal{F}_{Sub}$ as the first random permutation of $\mathcal{N}$ elements selected from $\mathcal{F}$ such that their sum is less than $\mathcal{S}$.
- Second, the original algorithm either finds a solution or terminates without success. We use an increasing sample size after each oversampling to reduce the error, and allow the solution to converge.

- Third, it is not sufficient for the elements in $\mathcal{F}_{Sub}$ to have a numerical sum close to the desired sum $\mathcal{S}$, but the distribution of the elements must also be close to the original distribution in $\mathcal{F}$. A goodness-of-fit test at the end of each oversampling step enforces this requirement. For our example, this ensures that the set of file sizes generated after resolving multiple constraints still follow the original distribution of file sizes.

The algorithm terminates successfully when the difference between the sums, and between the distributions, falls below the desired error levels. The success of the algorithm depends on the choice of the desired sum, and the *expected* sum (the sum due to the choice of parameters, *e.g.*, μ and σ); the farther the desired sum is from the expected sum, the lesser are the chances of success.

Consider an example where a user has specified a desired file system size of 90000 bytes, a lognormal file size distribution (μ=8.16, σ=2.46), and 1000 files. Figure 3.4(a) shows the convergence of the sum of file sizes in a sample set obtained with this distribution. Each line in the graph represents an independent trial, starting at a y-axis value equal to the sum of its initially sampled file sizes. Note that in this example, the initial sum differs from the desired sum by more than a 100% in several cases. The x-axis represents the number of extra iterations (*oversamples*) performed by the algorithm. For a trial to succeed, the sum of file sizes in the sample must converge to within 5% of the desired file system size. We find that in most cases λ ranges between 0 and 0.1 (*i.e.*, less than 10% oversampling); and in almost all cases, $\lambda \leq 1$.

The distribution of file sizes in $\mathcal{F}_{Sub}$ must be close to the original distribution in $\mathcal{F}$. Figure 3.4(b) and 3.4(c) show the difference between the original and constrained distributions for file sizes (for files by size, and files by bytes), for one successful trial from Figure 3.4(a). We choose these particular distributions as examples throughout this paper for two reasons. First, file size is an important parameter, so we want to be particularly thorough in its accuracy. Second, getting an accurate shape for the bimodal curve of files by bytes presents a challenge for Impressions; once we get our techniques to work for this curve, we are fairly confident of its accuracy on simpler distributions.

We find that Impressions resolves multiple constraints to satisfy the requirement on the sum, while respecting the original distributions. Table 3.4 gives the summary for the above example of file sizes for different values of the desired file system size. The expected sum of 1000 file sizes, sampled as specified in the table, is close to 60000. Impressions successfully converges the initial sample set to the desired sum with an average oversampling rate α less than 5%. The average difference between the desired and achieved sum β is close to 3%. The constrained distribution passes the two-sample K-S test at the 0.05 significance level, with the difference between the two distributions being fairly small (the D statistic of the K-S test is around 0.03, which represents the maximum difference between two empirical cumulative distributions).

We repeat the above experiment for two more choices of file system sizes, one lower than the expected mean (30K), and one higher (90K); we find that even when the desired sum is quite different from the expected sum, our algorithm performs well. Only for 2 of the 20 trials in the 90K case, did the algorithm fail to converge. For these extreme cases, we drop the initial sample and start over.

3.2.5 Interpolation and Extrapolation

Impressions requires knowledge of the distribution of file system parameters necessary to create a valid image. While it is tempting to imagine that Impressions has perfect knowledge about the nature of these distributions for all possible values and combinations of individual parameters, it is often impossible.

First, the empirical data is limited to what is observed in any given dataset and may not cover the entire range of possible values for all parameters. Second, even with an exhaustive dataset, the user may want to explore regions of parameter values for which no

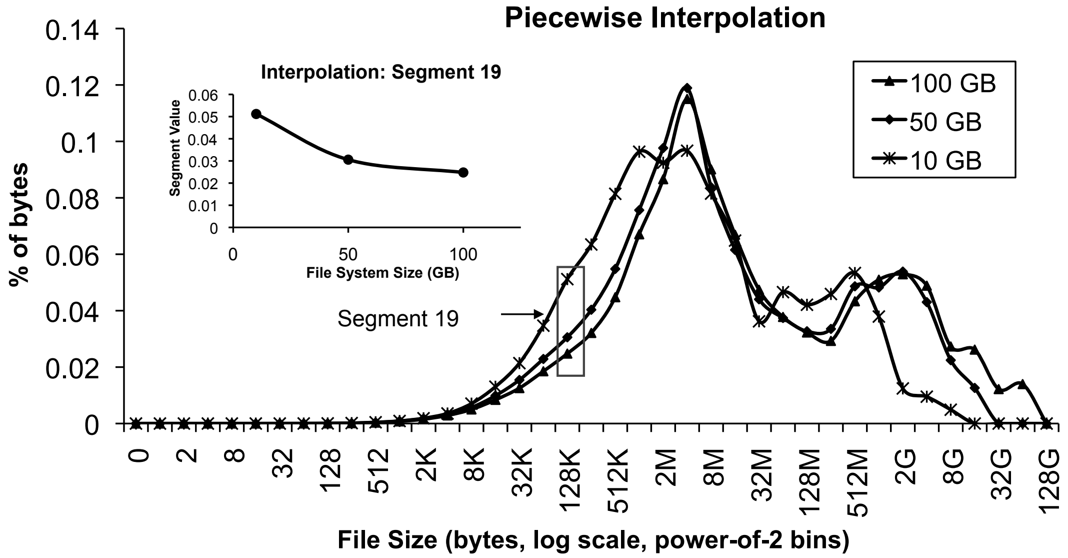


Figure 3.5 **Piecewise Interpolation of File Sizes.** *Piece-wise interpolation for the distribution of files with bytes, using file systems of 10 GB, 50 GB and 100 GB. Each power-of-two bin on the x-axis is treated as an individual* segment *for interpolation (inset). Final curve is the composite of all individual interpolated segments.*

data point exists, especially for "what if" style of analysis. Third, from an implementation perspective, it is more efficient to maintain compact representations of distributions for a few sample points, instead of large sets of data. Finally, if the empirical data is statistically insignificant, especially for outlying regions, it may not serve as an accurate representation. Impressions thus provides the capability for interpolation and extrapolation from available data and distributions.

Impressions needs to generate complete new curves from existing ones. To illustrate our procedure, we describe an example of creating an interpolated curve; extensions to extrapolation are straightforward. Figure 3.5 shows how Impressions uses *piece-wise interpolation* for the distribution of files with containing bytes. In this example, we start with

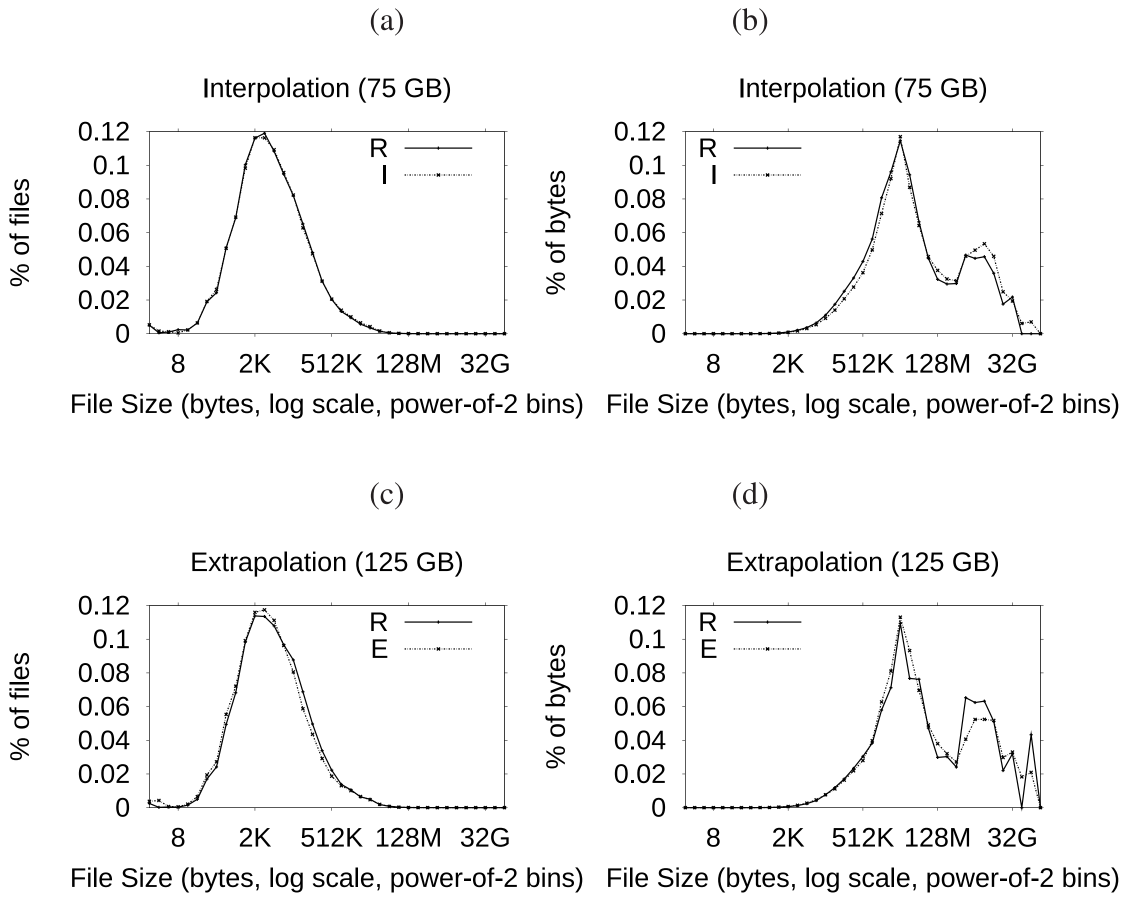


Figure 3.6 **Accuracy of Interpolation and Extrapolation.** *Shows results of applying piece–wise interpolation to generate file size distributions (by count and by bytes), for file systems of size 75 GB (a and b, respectively), and 125 GB (c and d, respectively).*

Distribution	FS Region (I/E)	D Statistic	K-S Test (0.05)
File sizes by count	75GB (I)	0.054	passed
File sizes by count	125GB (E)	0.081	passed
File sizes by bytes	75GB (I)	0.105	passed
File sizes by bytes	125GB (E)	0.105	passed

Table 3.5 **Accuracy of interpolation and extrapolation.** *Impressions produces accurate curves for file systems of size 75 GB and 125 GB, using interpolation (I) and extrapolation (E), respectively.*

the distribution of file sizes for file systems of size 10 GB, 50 GB and 100 GB, shown in the figure. Each power-of-two bin on the x-axis is treated as an individual *segment*, and the available data points within each segment are used as input for piece-wise interpolation; the process is repeated for all segments of the curve. Impressions combines the individual interpolated segments to obtain the complete interpolated curve.

To demonstrate the accuracy of our approach, we interpolate and extrapolate file size distributions for file systems of sizes 75 GB and 125 GB, respectively. Figure 3.6 shows the results of applying our technique, comparing the generated distributions with actual distributions for the file system sizes (we removed this data from the dataset used for interpolation). We find that the simpler curves such as Figure 3.6(a) and (c) are interpolated and extrapolated with good accuracy. Even for more challenging curves such as Figure 3.6(b) and (d), the results are accurate enough to be useful. Table 3.5 contains the results of conducting K-S tests to measure the goodness-of-fit of the generated curves. All the generated distributions passed the K-S test at the 0.05 significance level.

3.2.6 File Content

Actual file content can have substantial impact on the performance of an application. For example, Postmark [70], one of the most popular file system benchmarks, tries to simulate an email workload, yet it pays scant attention to the organization of the file system, and is completely oblivious of the file data. Postmark fills all the "email" files with the same data, generated using the same random seed. The evaluation results can range from misleading to completely inaccurate, for instance in the case of content-addressable storage (CAS). When evaluating a CAS-based system, the disk-block traffic and the corresponding performance will depend only on the unique content – in this case belonging to the largest file in the file system. Similarly, performance of Desktop Search and Word Processing applications is sensitive to file content.

In order to generate representative file content, Impressions supports a number of options. For human-readable files such as `.txt`, `.html` files, it can populate file content with random permutations of symbols and words, or with more sophisticated word-popularity models. Impressions maintains a list of the relative popularity of the most popular words in the English language, and a Monte Carlo simulation generates words for file content according to this model. However, the distribution of word popularity is heavy-tailed; hence, maintaining an exhaustive list of words slows down content generation. To improve performance, we use a word-length frequency model [129] to generate the long tail of words, and use the word-popularity model for the body alone.

According to the word-length frequency model the observed frequencies of word lengths is approximated by a variant of the gamma distribution, and is of the general form: $f_{exp} = a * L^b * c^L$, where f_{exp} is the observed frequency for word-length L, and (a,b,c) are language-specific parameters.

The user has the flexibility to select either one of the models in entirety, or a specific combination of the two. It is also relatively straightforward to add extensions in the future to generate more nuanced file content. An example of such an extension is one that carefully controls the degree of content similarity across files.

In order to generate content for typed files, Impressions either contains enough information to generate valid file headers and footers itself, or calls into a third-party library or software such as Id3v2 [96] for `mp3`; GraphApp [57] for `gif`, `jpeg` and other image files; Mplayer [93] for `mpeg` and other video files; asciidoc for `html`; and ascii2pdf for `PDF` files.

3.2.7 Disk Layout and Fragmentation

To isolate the effects of file system content, Impressions can measure the degree of on-disk fragmentation, and create file systems with user-defined degree of fragmentation. The extent of fragmentation is measured in terms of *layout score* [132]. A layout score of 1 means all files in the file system are laid out optimally on disk (*i.e.*, all blocks of any given file are laid out consecutively one after the other), while a layout score of 0 means that no two blocks of any file are adjacent to each other on disk.

Impressions achieves the desired degree of fragmentation by issuing pairs of temporary file create and delete operations, during creation of regular files. When experimenting with a file-system image, Impressions gives the user complete control to specify the overall layout score. In order to determine the on-disk layout of files, we rely on the information provided by debugfs. Thus currently we support layout measurement only for Ext2 and Ext3. In future work, we will consider several alternatives for retrieving file layout information across a wider range of file systems. On Linux, the FIBMAP and FIEMAP `ioctl()`s are available to map a logical block to a physical block [69]. Other file system-specific methods exist, such as the XFS_IOC_GETBMAP ioctl for XFS.

The previous approach however does not account for differences in fragmentation strategies across file systems. Impressions supports an alternate specification for the degree of fragmentation wherein it runs a pre-specified workload and reports the resulting layout score. Thus if a file system employs better strategies to avoid fragmentation, it is reflected in the final layout score after running the fragmentation workload.

There are several alternate techniques for inducing more realistic fragmentation in file systems. Factors such as burstiness of I/O traffic, out-of-order writes and inter-file layout are currently not accounted for; a companion tool to Impressions for carefully creating fragmented file systems will thus be a good candidate for future research.

FS distribution (Default)	Time taken (seconds) $Image_1$	$Image_2$
Directory structure	1.18	1.26
File sizes distribution	0.10	0.28
Popular extensions	0.05	0.13
File with depth	0.064	0.29
File and bytes with depth	0.25	0.70
File content (Single-word)	0.53	1.44
On-disk file/dir creation	437.80	1394.84
Total time	473.20 (8 mins)	1826.12 (30 mins)
File content (Hybrid model)	791.20	–
Layout score (0.98)	133.96	–

Table 3.6 **Performance of Impressions.** *Shows time taken to create file-system images with break down for individual features.* $Image_1$*: 4.55 GB,* 20000 *files,* 4000 *dirs.* $Image_2$*: 12.0 GB,* 52000 *files,* 4000 *dirs. Other parameters are default. The two entries for additional parameters are shown only for* $Image_1$ *and represent times in addition to default times.*

3.2.8 Performance

In building Impressions, our primary objective was to generate realistic file-system images, giving top priority to accuracy, instead of performance. Nonetheless, Impressions does perform reasonably well. Table 3.6 shows the breakdown of time taken to create a default file-system image of 4.55 GB. We also show time taken for some additional features such as using better file content, and creating a fragmented file system. Overall, we find that Impressions creates highly accurate file-system images in a reasonable amount of time and thus is useful in practice.

3.3 Case Study: Desktop Search

In this section, we use Impressions to evaluate desktop searching applications. Our goals for this case study are two-fold. First, we show how simple it is to use Impressions to create either representative images or images across which a single parameter is varied. Second, we show how future evaluations should report the settings of Impressions so that results can be easily reproduced.

We choose desktop search for our case study because its performance and storage requirements depend not only on the file system size and structure, but also on the type of files and the actual content within the files. We evaluate two desktop search applications: open-source Beagle [21] and Google's Desktop for Linux (GDL) [54]. Beagle supports a large number of file types using 52 search-filters; it provides several indexing options, trading performance and index size with the quality and feature-richness of the index. Google Desktop does not provide as many options: a web interface allows users to select or exclude types of files and folder locations for searching, but does not provide any control over the type and quality of indexing.

3.3.1 Representative Images

Developers of data-intensive applications frequently need to make assumptions about the properties of file-system images. For example, file systems and applications can often be optimized if they know properties such as the relative proportion of meta-data to

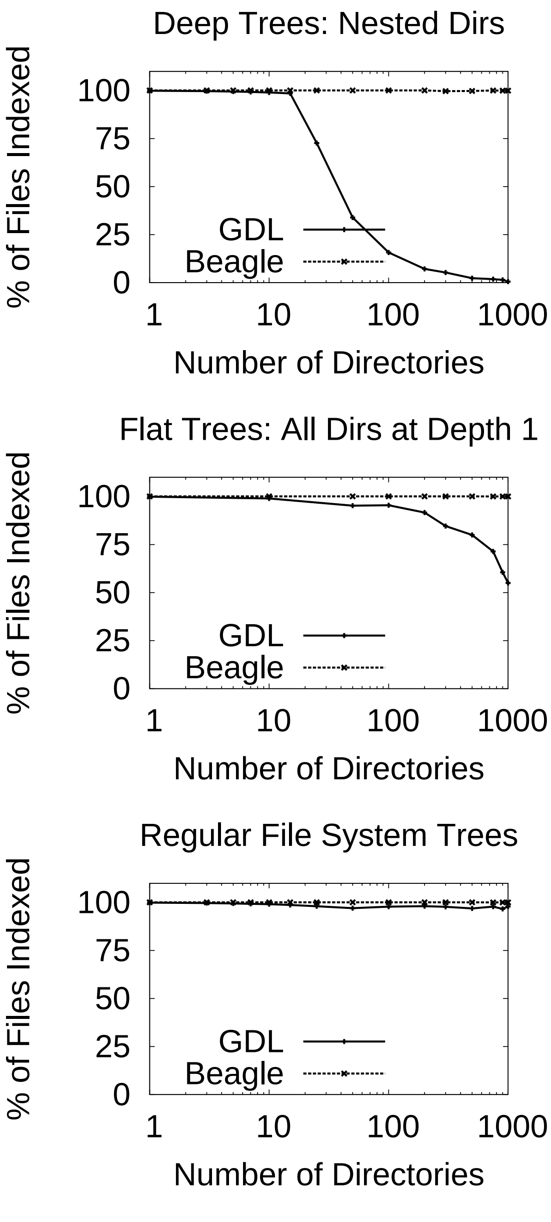


Figure 3.7 **Tree Depth and Completeness of Index.** *Shows the percentage of files indexed by Beagle and GDL with varying directory tree depths in a given file-system image*

App	Parameter & Value	Comment on Validity
GDL	File content $<$ 10 deep	10% of files and 5% of bytes $>$ 10 deep (content in deeper namespace is growing)
GDL	Text file sizes $<$ 200 KB	13% of files and 90% of bytes $>$ 200 KB
Beagle	Text file cutoff $<$ 5 MB	0.13% of files and 71% of bytes $>$ 5 MB
Beagle	Archive files $<$ 10 MB	4% of files and 84% of bytes $>$ 10 MB
Beagle	Shell scripts $<$ 20 KB	20% of files and 89% of bytes $>$ 20 KB

Figure 3.8 **Debunking Application Assumptions.** *Examples of assumptions made by Beagle and GDL, along with details of the amount of file-system content that is not indexed as a consequence.*

data in representative file systems. Previously, developers could infer these numbers from published papers [8, 41, 120, 128], but only with considerable effort. With Impressions, developers can simply create a sample of representative images and directly measure the properties of interest.

Table 3.8 lists assumptions we found in GDL and Beagle limiting the search indexing to partial regions of the file system. However, for the representative file systems in our data set, these assumptions omit large portions of the file system. For example, GDL limits its index to only those files less than ten directories deep; our analysis of typical file systems indicates that this restriction causes 10% of all files to be missed.

Figure 3.7 shows one such example: it compares the percentage of files indexed by Beagle and GDL for a set of file-system images. The topmost graph shows the results for *deep* file-system trees created by successively nesting a new directory in the parent directory; a file system with D directories will thus have a maximum depth of D. The y-axis shows the % of files indexed, and the x-axis shows the number of directories in the file system. We find that GDL stops indexing content after depth 10, while Beagle indexes 100% of the files. The middle graph repeats the experiment on flat trees, with all directories at depth 1. This time, GDL's percentage completeness drops off once the number of directories exceeds 10. For regular file system trees, shown in the lowermost graph, we find that both Beagle and GDL achieve near 100% completeness. Since the percentage of user-generated content deeper in the namespace is growing over the years, it might be useful to design search indexing schemes which are better suited for deeper name spaces.

This strange behavior further motivates the need for a tool like Impressions to be a part of any application designer's toolkit. We believe that instead of arbitrarily specifying hard values, application designers should experiment with Impressions to find acceptable choices for representative images.

We note that Impressions is useful for discovering these application assumptions and for isolating performance anomalies that depend on the file-system image. Isolating the impact of different file system features is easy using Impressions: evaluators can use Impressions to create file-system images in which only a single parameter is varied, while all other characteristics are carefully controlled.

This type of discovery is clearly useful when one is using closed-source code, such as GDL. For example, we discovered the GDL limitations by constructing file-system images across which a single parameter is varied (*e.g.*, file depth and file size), measuring the percentage of indexed files, and noticing precipitous drops in this percentage. This type of controlled experimentation is also useful for finding non-obvious performance interactions in open-source code. For instance, Beagle uses the *inotify* mechanism [68] to track each directory for change; since the default Linux kernel provides 8192 watches, Beagle resorts to manually crawling the directories once their count exceeds 8192. This deterioration in performance can be easily found by creating file-system images with varying numbers of directories.

3.3.2 Reproducible Images

The time spent by desktop search applications to crawl a file-system image is significant (*i.e.*, hours to days); therefore, it is likely that different developers will innovate in this area. In order for developers to be able to compare their results, they must be able to ensure they are using the same file-system images. Impressions allows one to precisely control the image and report the parameters so that the exact same image can be reproduced.

For desktop search, the type of files (*i.e.*, their extensions) and the content of files has a significant impact on the time to build the index and its size. We imagine a scenario in which the Beagle and GDL developers wish to compare index sizes. To make a meaningful comparison, the developers must clearly specify the file-system image used; this can be done easily with Impressions by reporting the size of the image, the distributions listed in Table 3.2, the word model, disk layout, and the random seed. We anticipate that most benchmarking will be done using mostly default values, reducing the number of Impressions parameters that must be specified.

An example of the reporting needed for reproducible results is shown in Figure 3.9. In these experiments, all distributions of the file system are kept constant, but only either text files (containing either a single word or with the default word model) or binary files are created. These experiments illustrate the point that file content significantly affects the index size; if two systems are compared using different file content, obviously the results are meaningless. Specifically, different file types change even the relative ordering of index size between Beagle and GDL: given text files, Beagle creates a larger index; given binary files, GDL creates a larger index.

Figures 3.10 gives an additional example of reporting Impressions parameters to make results reproducible. In these experiments, we discuss a scenario in which different developers have optimized Beagle and wish to meaningfully compare their results. In this

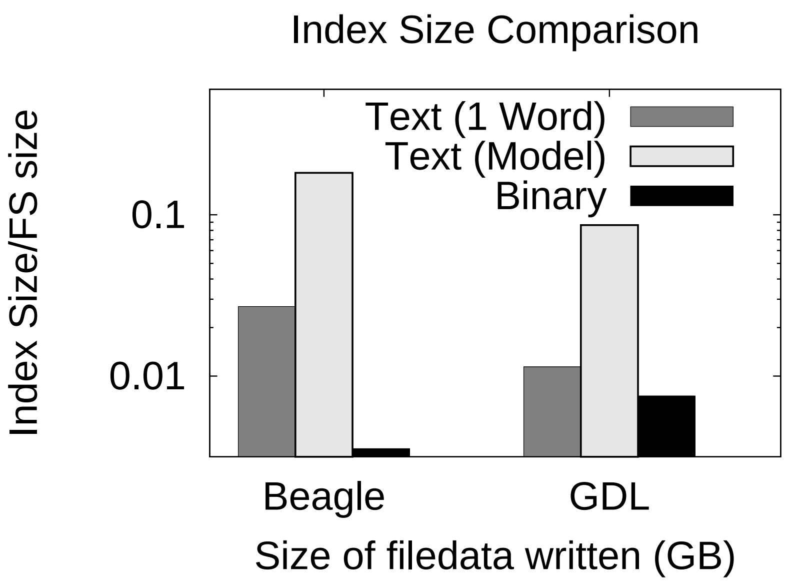
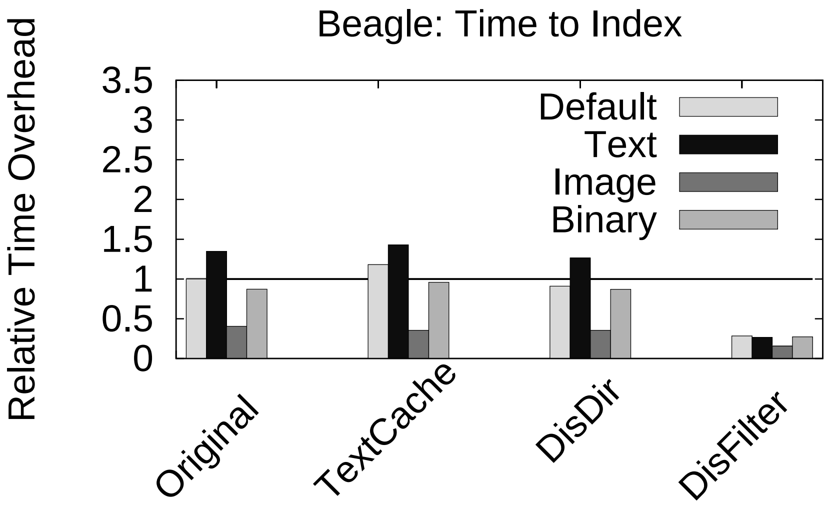


Figure 3.9 **Impact of file content.** *Compares Beagle and GDL index time and space for wordmodels and binary files. Google has a smaller index for wordmodels, but larger for binary. Uses Impressions default settings, with FS size 4.55 GB, 20000 files, 4000 dirs.*

scenario, the original Beagle developers reported results for four different images: the default, one with only text files, one with only image files, and one with only binary files. Other developers later create variants of Beagle: *TextCache* to display a small portion of every file alongside a search hit, *DisDir* to disable directory indexing, and *DisFilter* to index only attributes. Given the reported Impressions parameters, the variants of Beagle can be meaningfully compared to one another.

In summary, Impressions makes it extremely easy to create both controlled and representative file-system images. Through this brief case study evaluating desktop search applications, we have shown some of the advantages of using Impressions. First, Impressions enables developers to tune their systems to the file system characteristics likely to be found in their target user populations. Second, it enables developers to easily create images where one parameter is varied and all others are carefully controlled; this allows one to



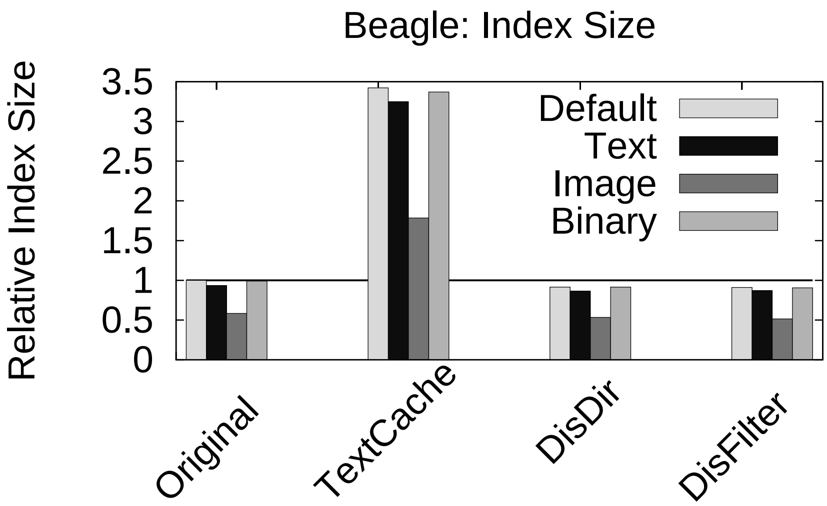


Figure 3.10 **Reproducible images: impact of content.** *Using Impressions to make results reproducible for benchmarking search. Vertical bars represent file systems created with file content as labeled. The* Default *file system is created using Impressions default settings, and file system size 4.55 GB, 20000 files, 4000 dirs. Index options: Original – default Beagle index. TextCache – build text-cache of documents used for snippets. DisDir – don't add directories to the index. DisFilter – disable all filtering of files, only index attributes.*

assess the impact of a single parameter. Finally, Impressions enables different developers to ensure they are all comparing the same image; by reporting Impressions parameters, one can ensure that benchmarking results are reproducible.

3.4 Other Applications

Besides its use in conducting representative and reproducible benchmarking, Impressions can also be handy in other experimental scenarios. In this section we present two examples, the usefulness of Impressions in generating realistic rules of thumb, and in testing soundness of hypothesis.

3.4.1 Generating Realistic Rules of Thumb

In spite of the availability of Impressions, designers of file systems and related software will continue to rely on rules of thumb to make design decisions. Instead of relying on old wisdom, one can use Impressions to generate realistic rules of thumb. One example of such a rule of thumb is to calculate the overhead of file-system metadata – a piece of information often needed to compute the cost of different replication, parity or check summing schemes for data reliability. Figure 3.11 shows the percentage of space taken by metadata in a file system, as we vary the distribution of file sizes. We find that the overhead can vary between 2 and 14% across the file size distributions in this example. Similarly, Impressions can be used to compute other rules of thumb for different metadata properties.

3.4.2 Testing Hypothesis

In our experience, we found Impressions convenient and simple to use for testing hypothesis regarding application and file system behavior, hiding away the statistical complexity of the experiment from the end-user. To illustrate this, we describe our experience with a *failed* experiment.

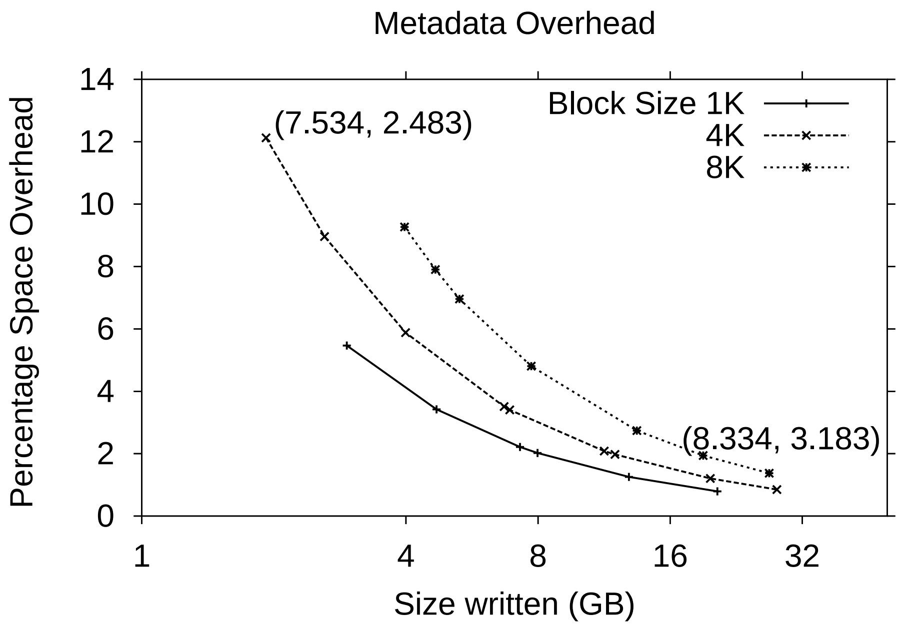


Figure 3.11 **Metadata Overhead.** *Shows the relative space overhead of file-system metadata with varying file-size distribution, modeled by (μ, σ) parameters of a lognormal distribution (shown in parentheses for the two extremes).*

It was our hypothesis that the distribution of bytes and files by namespace depth would affect the time taken to build the search index: indexing file content in deeper namespace would be slower. To test our hypothesis, all we had to do was use Impressions to create file-system images, and measure the time taken by Beagle to build the index, varying only a single parameter in the configuration file for each trial: the λ value governing the Poisson distribution for file depth. Although our hypothesis was not validated by the results (*i.e.*, we didn't find significant variation in indexing time with depth), we found Impressions to be suitable and easy to use for such experimentation.

3.5 Related Work

We discuss previous research in three related areas. First, we discuss existing tools for generating file-system images; second, we present prior research on improving file system benchmarking; finally, we discuss existing models for explaining file system metadata properties.

3.5.1 Tools for Generating File-System Images

We are not aware of any existing system that generates file-system images with the level of detail that Impressions delivers; here we discuss some tools that we believe provide some subset of features supported by Impressions.

FileBench, a file system workload framework for measuring and comparing file system performance [112] is perhaps the closest to Impressions in terms of flexibility and attention to detail. FileBench generates test file system images with support for different directory hierarchies with namespace depth and file sizes according to statistical distributions. We believe Impressions includes all the features provided by FileBench and provides additional capabilities; in particular, Impressions allows one to contribute newer datasets and makes it easier to plug in distributions. FileBench also does not provide support for allowing user-specified constraints.

The SynRGen file reference generator by Ebling and Satyanarayan [44] generates synthetic equivalents for real file system users. The *volumes* or images in their work make use of simplistic assumptions about the file system distributions as their focus is on user access patterns.

File system and application developers in the open-source community also require file-system images to test and benchmark their systems, tools for which are developed in-house, often customized to the specific needs of the system being developed.

Genbackupdata is one such tool that generates test data sets for performance testing of backup software [155]. Like Impressions, but in a much simplified fashion, it creates a directory tree with files of different sizes. Since the tool is specifically designed for backup applications, the total file system size and the minimum and maximum limits for file sizes are configurable, but not the file size distribution or other aspects of the file system. The program can also modify an existing directory tree by creating new files, and deleting, renaming, or modifying existing files, inducing fragmentation on disk.

Another benchmarking system that generates test file systems matching a specific profile is Fstress [12]. However, it does contain many of the features found standard in Impressions, such as popularity of file extensions and file content generation according to file types, supporting user-specified distributions for file system parameters and allowing arbitrary constraints to be specified on those parameters.

3.5.2 Tools and Techniques for Improving Benchmarking

A number of tools and techniques have been proposed to improve the state of the art of file and storage system benchmarking. Chen and Patterson proposed a "self-scaling" benchmark that scales with the I/O system being evaluated, to stress the system in meaningful ways [32]. Although useful for disk and I/O systems, the self-scaling benchmarks are not directly applicable for file systems.

TBBT is a NFS trace replay tool that derives the file-system image underlying a trace [162]. It extracts the file system hierarchy from a given trace in depth-first order and uses that during initialization for a subsequent trace replay. While this ensures a consistent file-system image for replay, it does not solve the more general problem of creating accurately controlled images for all types of file system benchmarking.

The Auto-Pilot tool [159] provides an infrastructure for running tests and analysis tools to automate the benchmarking process. Auto-Pilot can help run benchmarks with relative ease by automating the repetitive tasks of running, measuring, and analyzing a program through test scripts.

3.5.3 Models for File-System Metadata

Several models have been proposed to explain observed file-system phenomena. Mitzenmacher proposed a generative model, called the Recursive Forest File model [90] to explain the behavior of file size distributions. The model is dynamic as it allows for the creation of new files and deletion of old files. The model accounts for the hybrid distribution of file sizes with a lognormal body and Pareto tail.

Downey's Multiplicative File Size model [43] is based on the assumption that new files are created by using older files as templates e.g., by copying, editing or filtering an old file. The size of the new file in this model is given by the size of the old file multiplied by an independent factor.

The HOT (Highly Optimized Tolerance) model provides an alternate generative model for file size distributions. These models provide an intuitive understanding of the underlying phenomena, and are also easier for computer simulation. In future, Impressions can be enhanced by incorporating more such models.

3.6 Conclusion

File system benchmarking is in a state of disarray. One key aspect of this problem is generating realistic file-system state, with due emphasis given to file-system metadata and file content. To address this problem, we have developed Impressions, a statistical framework to generate realistic and configurable file-system images. Impressions provides the user flexibility in selecting a comprehensive set of file system parameters, while seamlessly ensuring accuracy of the underlying images, serving as a useful platform for benchmarking.

In our experience, we find Impressions easy to use and well suited for a number of tasks. It enables application developers to evaluate and tune their systems for realistic file system characteristics, representative of target usage scenarios. Impressions also makes it feasible to compare the performance of systems by standardizing and reporting all used parameters, a requirement necessary for benchmarking. We believe Impressions will prove to be a valuable tool for system developers and users alike; we have made it publicly available for download. Please visit the URL `http://www.cs.wisc.edu/adsl/Software/Impressions/` to obtain a copy.

Chapter 4

Practical Storage System Benchmarking with *Compressions*

File and storage systems are currently difficult to benchmark. So far we have discussed two important challenges in file-system benchmarking: recreating benchmarking state representative of real-world conditions, and ensuring reproducibility of the benchmarking state to allow fair comparison. The time and effort required to ensure that the above conditions are met often discourages developers from using benchmarks that matter, settling instead for the ones that are easy to set up and use. These deficiencies in benchmarking point to a thematic problem – when it comes to actual usage, ease of use and practicality often overshadow realism and accuracy.

In practice, realistic benchmarks (and realistic configurations of such benchmarks) tend to be much larger and more complex to set up than their trivial counterparts. File system traces (*e.g.*, from HP Labs [113]) are good examples of such workloads, often being large and unwieldy. In many cases the evaluator has access to only a modest storage capacity, making it harder still to employ large, real workloads.

Two trends further exacerbate the problems in file and storage benchmarking. First, storage capacities have seen a tremendous increase in the past few years; Terabyte-sized disks are now easily available for desktop computers; enterprise systems are now frequently working with Petabyte-scale storage. The problem with using large file-system images for experimentation is that creating and running workloads on them can be time consuming. Second, real applications and benchmarks that developers and evaluators care about are taking increasingly longer to run. Examples of such applications include file-system integrity checkers like `fsck`, desktop search indexing, and backup software, taking anywhere from several hours to a few days to run on a Terabyte-sized partition.

Benchmarking with such applications on large storage devices is a frequent source of frustration for file-system evaluators; the scale alone acts as a strong deterrent against using larger albeit realistic benchmarks [146]. Given the rate at which storage capacities are increasing, running toy workloads on small disks is no longer a satisfactory alternative. One obvious solution is to continually upgrade one's storage capacity. However, this is an expensive, and perhaps an infeasible solution, especially to justify the costs and administrative overheads solely for benchmarking.

In order to encourage developers of file systems and related software to adopt larger, more realistic benchmarks and configurations, we need means to make them practical to run on modest storage infrastructure. To address this problem, we have developed Compressions, a "scale down" benchmarking system that allows one to run large, complex workloads using relatively small storage capacities by *scaling down* the storage requirements transparent to the workload. Compressions makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system.

Our observation is that in many cases, the user does not care about the contents of individual files, but only about the structure and properties of the metadata that is being stored on disk. In particular, for the purposes of benchmarking, many applications do not write or read file contents at all (*e.g.*, `fsck`); the ones that do, often do not care what the contents are as long as *some* valid content is made available (*e.g.*, a backup software). Since file data constitutes a significant fraction of the total file system size, ranging anywhere from 90 to 99% depending on the actual file-system image (Chapter 2), avoiding the need to store file data has the potential to save a lot of time and storage space during benchmarking.

The key idea in Compressions is to create a "compressed" version of the original file-system image for the purposes of benchmarking. In the compressed image, unneeded user data blocks are omitted and file system metadata blocks (e.g., inodes, directories and indirect blocks) are laid out more efficiently on disk; in the simplest case, metadata blocks are written out consecutively from the beginning of the disk. To ensure that applications and benchmark workloads remain unaware of this interposition, whenever necessary, Compressions synthetically produces file data using a suitably modified version of Impressions; metadata reads and writes are redirected and accessed appropriately. Compressions uses an in-kernel model of the disk and storage stack to determine the runtime of the benchmark workload on the original uncompressed image. The storage model calculates the run times of all individual requests as they would have executed on the uncompressed image.

In our evaluation of Compressions with workloads like PostMark, `mkfs` (a tool to build a file system on a storage device) and other microbenchmarks, we find that Compressions delivers on its promise and reduces the required storage size and the runtime. Depending on the workload and the underlying file-system image, the size of the compressed image can range anywhere from 1 to 10% of the original, a huge reduction in the required disk size for benchmarking. Compressions also reduces the time taken to run the benchmark by avoiding a significant fraction of disk I/O and disk seeks. The storage model within Compressions is fairly accurate in spite of operating in real-time, and imposes an almost negligible overhead on the workload execution.

Compressions supports two modes of operation – in the first mode, the storage model returns instantaneously after computing the time taken to run the benchmark workload on the uncompressed disk; in the second mode, Compressions models the runtime and introduces an appropriate delay before returning to the application. Compressions thus allows one to run benchmark workloads that require file-system images orders of magnitude larger than the available disk, and to run much faster than usual if needed, all this while still reporting the runtime as it would have taken on the original image; we believe Compressions provides a practical approach to run large, real workloads with a modest overhead and virtually no extra expense in frequently upgrading storage infrastructure for benchmarking.

In this chapter we present the design, implementation and evaluation of the Compressions benchmarking system. We start with a background on the storage stack and modeling storage systems in §4.1 and then present the design details of Compressions in §4.2; the storage model developed for Compressions is discussed separately in §4.3. We present the evaluation results for Compressions in §4.4, discuss related research in §4.5, and conclude in §4.6.

4.1 Background

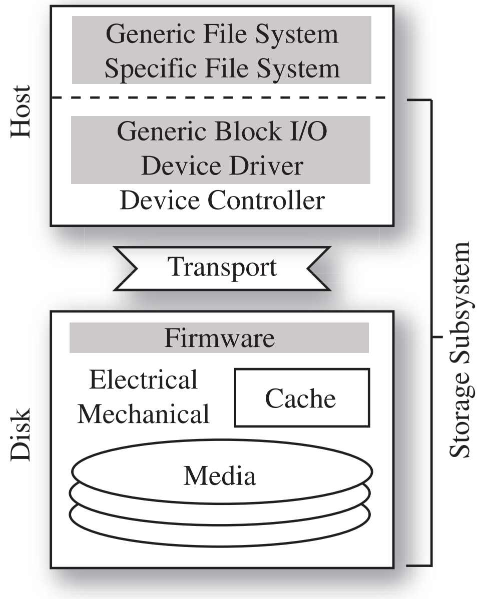


Figure 4.1 **The storage stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

This section provides a background on the components of the storage stack and how to model each of them. We first provide a brief overview of the storage stack in a computer system, and then present an overview on modeling disk drives and the storage stack.

4.1.1 Storage Stack

A storage stack consists of many different layers each providing an abstraction of the layer beneath to the layer above. Figure 4.1 shows the storage stack in a typical computer system with the disk drive at the bottom of the stack. The drive can be a traditional rotating hard disk, or a solid-state disk [10]. A hard disk contains magnetic storage media which stores the data and numerous other electrical and mechanical components to provide access to the media for reading and writing data. Most disks also have a small amount of on-board cache serving as a buffer for writes and for prefetching reads. Solid-state disks (SSDs) are constructed using some form of cell-based non-volatile memory such as NAND or NOR flash as the storage media. In addition, SSDs also contain some amount of RAM and a controller to maintain logical to physical mappings and process requests.

Another important component present in disk drives is the firmware: complex embedded code to control and manage the disk, and provide higher-level functions. A transport medium connects the drive to the host. SCSI, IDE and SATA are some common forms of bus transport protocols.

At the host, a hardware device controller provides a communication pathway to the external device. Higher up in the storage hierarchy are software components starting with the device driver that controls the hardware. The file system and the generic block I/O layer form the next set of layers. The generic block I/O layer provides functionality common to all (or several) file systems, such as prefetching, block reordering and even some error handling. The file system sits on top of the block I/O layer, managing its internal data structures and providing specific functionalities. A generic file system layer is commonly used to provide a standard interface to the applications using the file system (*e.g.*, POSIX). The generic file system layer maps generic operations to file system specific operations through another standardized interface (*e.g.*, Vnode/VFS [73]).

4.1.2 Storage Systems

When evaluators wish to benchmark a system for which a prototype does not exist or one that is otherwise difficult to obtain and set up, a model of the system can be used instead. In order to model any system we must first understand its behavior. In this section we present a brief overview of storage systems . Since the focus of this dissertation is on evaluating performance and not other aspects of a system such as reliability, the discussions on modeling a storage system are geared towards the components and features that are crucial for measuring performance. We start with a primer on disk drives and then briefly discuss the remainder of the storage stack.

4.1.2.1 Disk Drives

Modern disk drives are extremely complex; modeling a disk drive requires a thorough understanding of its internal structure and interactions between various components. A typical hard disk consists of magnetic media, electrical and mechanical components. In addition, most disks have a small amount of memory, and firmware to manage the disk. A hard disk contains one or more *platters* of magnetically coated media to record data, with each platter having two surfaces to hold data. A dedicated read/write head is provisioned for each surface of a platter. An arm assembly moves the read/write heads into position for individual request, with an actuator servo mechanism providing precise control over the placement of the head.

Data is laid out on the platter in concentric circular tracks. A single platter may contain tens of thousands of tracks, each further subdivided into *sectors*, the smallest addressable unit of data storage. Sectors are usually 512 bytes in size, however their real physical size is sometimes expanded (520 bytes) to provide some extra space to store error correcting codes (ECC). The outer tracks in modern disk drives are denser and therefore have higher transfer rate than the inner tracks. This phenomenon is often called disk *zoning* and is utilized to improve performance by laying out frequently accessed data on the outer tracks.

To the host system the disk appears as a linear array of addressable blocks. The file system can choose the block size at the time of volume creation and is usually between 512 and 4096 bytes. Each individual block is identified by a logical address (block number) and the disk internally maintains a mapping to corresponding disk sectors.

4.1.2.2 Storage Stack

The disk drive is perhaps the single most complex entity to model in the storage stack but other components of the stack also need to be modeled for overall accuracy. Important amongst them from a performance perspective are the transport and the request queues at each layer in the storage stack.

The transport is usually a bus protocol for lower-end systems (*e.g.*, ATA or SCSI), whereas networks are common in higher-end systems (*e.g.*, FibreChannel). Modeling the transport requires modeling the bus transfer bandwidth and its command and communication protocol.

Each block device driver in a system also maintains a *request queue* that contains the list of all requests pending submission to the corresponding device. The request queues act as containers for performing I/O scheduling operations like request re-ordering and request merging to improve disk performance. A newly created I/O request is not serviced immediately but instead gets added to the appropriate request queue.

4.2 The Compressions Framework

In this section we present the design and implementation of Compressions: a benchmarking system that makes it practical to run large, real workloads using a relatively modest storage capacity. We first present the design goals for Compressions, then discuss its overall architecture, and finally describe in detail all its constituent subsystems.

4.2.1 Design Goals

The design of Compressions is influenced by the following goals:

- **Scalability:** Running large, real benchmarks with a modest storage capacity requires Compressions to maintain additional data structures and mimic several operations on its own; our goal is to ensure that Compressions works well as disk capacities scale.

- **Accuracy:** Compressions needs to model a large number of I/Os concurrent with workload execution; our goal is to be able to accurately predict application runtime without slowing the system down.

- **No application modification:** In building Compressions our goal is to avoid the need to change applications or benchmarks. A benchmark workload should be able to run unmodified on Compressions without knowing that the storage system underneath has significantly less capacity than advertised.

- **Easy to use:** In our experience with developing and benchmarking file system and storage technologies we have observed that the importance of ease of use is often under-estimated. In designing Compressions, one of our goals is to provide an intuitive, easy to use interface to the user.

- **Speedier benchmarking:** System design is often an iterative process and benchmarking the systems can be time consuming. If the application under test does not have strict timing dependencies, Compressions can speedup workload execution if desired and act as a catalyst for benchmarking.

4.2.2 Basic Architecture

Compressions consists of four primary components: a mechanism for block classification and elimination of writes to data blocks; the MetaMap for remapping metadata blocks, the DataGenerator for generating synthetic file content, and the Storage Model to compute the actual runtime of an application on a storage system of choice.

The different components of Compressions are used in tandem to allow large workloads to run on a storage system with limited capacity. First, Compressions intercepts and discards all non-essential writes (*i.e.*, writes to file data blocks) to reduce the amount of storage space and I/O traffic to the disk. Second, a small fraction of writes to essential blocks (*i.e.*, metadata blocks) are passed on to the real device. These writes are suitably remapped so as to lay them out more efficiently on disk, requiring a smaller storage capacity. Third, in order to deliver accurate timing statistics, a model of the underlying storage stack and disk is maintained. Finally, synthetic content is generated in order to service read requests to data blocks that were previously discarded.

Figure 4.2 shows the basic architecture of Compressions and its placement in the storage stack as a software layer directly beneath the file system and above the storage disk (*i.e.*, as a pseudo-device). This pseudo-device appears as a regular disk to the file system and interposes on all I/O requests to the real storage device.

The pseudo-device driver is responsible for classifying blocks addressed in a request as data or metadata and preventing I/O requests to data blocks from going to the real disk (*i.e.*, squashing). The pseudo-device driver intercepts all writes to data blocks, records the block address if necessary, and discards the actual write. I/O requests to metadata blocks are passed on to the MetaMap.

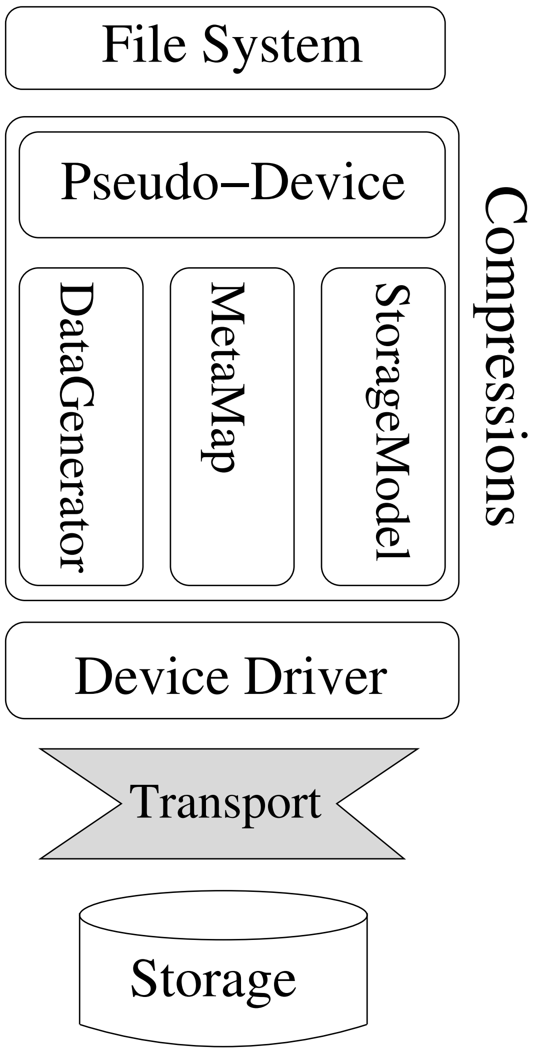


Figure 4.2 **Compressions Overview.** *The figure presents the architecture of Compressions.*

The MetaMap module is responsible for laying out metadata blocks more efficiently on the disk. It intercepts all write requests to metadata blocks, generates a remapping for the set of blocks addressed in the request, and writes out the metadata blocks to the remapped locations. The remapping is stored in the MetaMap to service subsequent reads to these metadata blocks.

By performing the above mentioned tasks, the pseudo-device driver and the MetaMap modify the original I/O request stream by altering the location of metadata blocks on disk and discarding data blocks altogether. These modifications in the disk traffic substantially change the application runtime, rendering it less useful for benchmarking. The Storage Model in Compressions provides an extrication from this predicament by carefully simulating a potentially different storage subsystem underneath to model the time taken to run. By doing so in an online fashion with little overhead, the Storage Model makes it feasible to run large workloads in a space and time-efficient manner.

Writes to data blocks are not saved on disk, but reads to these blocks could still be issued by the application; in order to allow applications to run transparently, the DataGenerator is responsible for generating synthetic content to service subsequent reads to data blocks that were written earlier during benchmarking and discarded. The DataGenerator contains a number of built-in schemes to generate different kinds of content and also allows the application to provide hints to generate more tailored content.

The details of the individual components are discussed next in this section, except the disk model, which is discussed separately in Section 4.3.

4.2.3 Block Classification and Data Squashing

One of the primary requirements for Compressions to operate is the ability to classify a block as metadata or data. Compressions leverages prior work on Semantically-smart

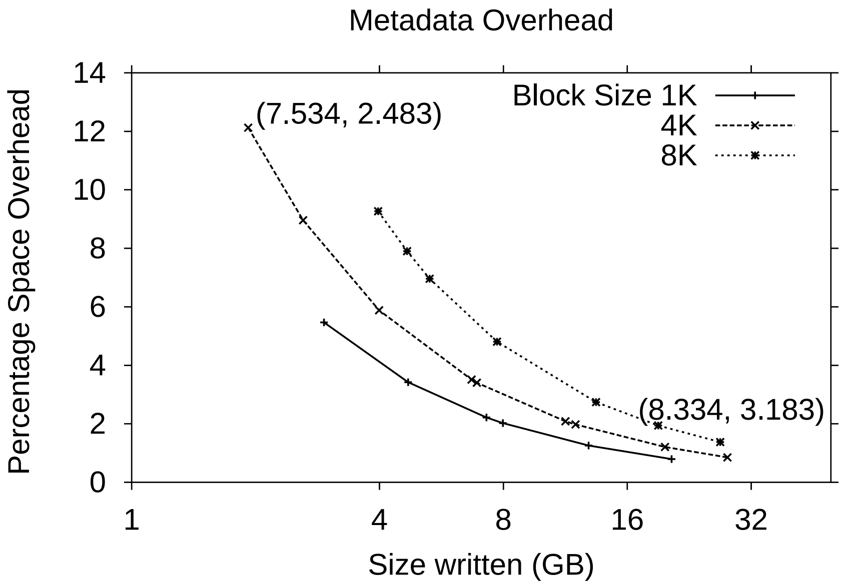


Figure 4.3 **Metadata Overhead.** *Shows the relative space overhead of file-system metadata with varying file-size distribution, modeled by (μ, σ) parameters of a lognormal distribution (shown in parentheses for the two extremes).*

Disk Systems [130] to implement block classification inside the pseudo-device driver; each block read or written to the pseudo-device can be classified as data or metadata.

For file systems belonging to the FFS family, such as Ext2 and Ext3, the majority of the blocks are statically assigned for a given file system size and configuration at the time of file system creation; the allocation for the statically assigned blocks doesn't change during the lifetime of the file system. Blocks that fall in this category include the super block, group descriptors, inode and data bitmaps, inode blocks and blocks belonging to the journal. Dynamically allocated blocks include directory, indirect (single, double, or triple indirect) and data blocks. Unless all blocks contain some self-identification information, in order to accurately classify a dynamically allocated block, the system needs to track the inode that points to the particular block and infer its current status. Compressions tracks writes to inode blocks, inode bitmaps and data bitmaps to infer when a file or directory is allocated and deallocated; it uses the content of the inode to enumerate the indirect and directory blocks that belong to a particular file or directory. This classification is used subsequently to either squash or remap a given block in a *type-aware* fashion [106] based on its current status as a data or metadata block.

There are two benefits to data squashing. First, the total storage capacity required in absence of data blocks is substantially lower than before. Second, avoiding writes to data blocks reduces the number of disk I/Os and correspondingly the disk seeks, improving the runtime.

Figure 4.3 shows the relative ratio of metadata to data blocks with varying file-size distributions. As seen in the figure, file systems typically have a much higher percentage of data blocks as compared to metadata, ranging anywhere from 90 to 99%; the higher the ratio of data writes, the greater the advantages of data squashing. Having such a high fraction of blocks as data makes data squashing an especially attractive feature to have for benchmarking applications where file contents are not read subsequent to being written. A good example of such an application is a file system integrity checker such as `fsck`.

4.2.3.1 Data Cache

In Compressions the classification of dynamically allocated blocks depends on observing a write to the corresponding inode entry. It is often the case that the blocks pointed to by an inode are written out before the corresponding inode block; if a classification attempt is made at this time, an indirect or directory block will be misclassified as an ordinary data block. Eventually the inode block is written, such as due to a periodic flush of the buffer cache, and the misclassification can be rectified. This is unacceptable for Compressions since a transient error leads to the "data" block being discarded prematurely and could cause irreparable damage to the file system. For example, if a directory or indirect block is accidentally discarded, it could lead to file system corruption.

To rectify this problem, Compressions temporarily buffers all data blocks in the *Data Cache* until an accurate classification can be made or a pre-specified time quanta expires, whichever happens first. The design of the Data Cache is relatively straightforward – all dynamically allocated blocks for which a corresponding inode entry has not yet been written are added to the Data Cache. When an inode does get written, blocks that are classified as directory or indirect are passed on to the MetaMap for remapping and are written out to persistent storage, whereas blocks classified as data are discarded at that time; all entries corresponding to that inode are then removed. To prevent data blocks from indefinitely occupying space, the Data Cache periodically cleans out data block entries for which no inode write has been observed for some time; this time quanta is usually set equal to or greater than the frequency with which contents of the file system buffer cache are flushed out to disk which is typically less than a minute.

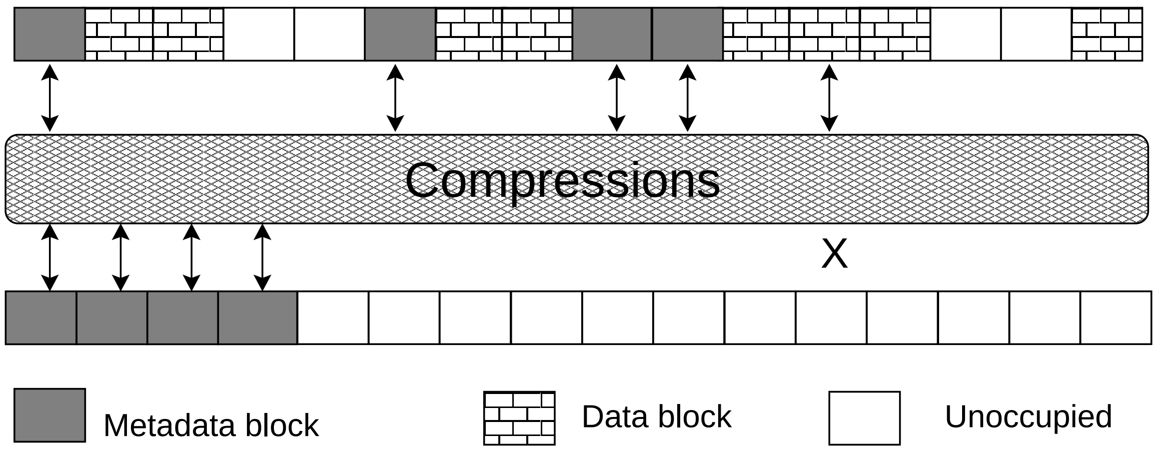


Figure 4.4 **Metadata Remapping and Data Squashing.** *The figure presents how metadata gets remapped and data blocks are squshed in Compressions. The disk image above Compressions is the one visible to the application and the one below it depicts the compressed disk image.*

4.2.4 Metadata Remapping

With data squashing turned on, disk I/O is issued only for metadata blocks and leads to sparse block allocation over the entire disk. In order to make efficient use of the available storage, Compressions reclaims the space meant for the (unused) data block locations through the MetaMap, wherein metadata blocks are remapped to a contiguous region of the disk, typically starting at the beginning of the disk partition. Figure 4.4 shows how Compressions makes use of metadata remapping to free up a large percentage of the required disk space; a much smaller disk can now service the requests of the benchmark workload.

Compressions creates a remap entry for each metadata block (*e.g.*, super block, indirect block etc) or range of metadata blocks (*e.g.*, group descriptors, inode block table etc) in the MetaMap; by allowing an arbitrary range of blocks to be remapped together, the MetaMap provides an efficient translation service for block allocation, lookup and deallocation. Range remapping also preserves sequentiality of the blocks on disk. In addition, a *remap bitmap* is maintained to keep track of block allocation and deallocation for the compressed disk image represented by the MetaMap; the remap bitmap supports allocation both of a single remapped block and a range of remapped blocks. For the Ext3 file system, since most of the blocks are statically allocated, the remapping for these blocks can also be done statically. Subsequent writes to other metadata blocks are remapped dynamically; when metadata blocks are deallocated, corresponding entries from the MetaMap and the remap bitmap are removed.

The MetaMap thus has the following two advantages. First, by compacting the sparsely allocated blocks, a large portion of the disk space is freed up, paving the way for a smaller disk to suffice the requirements. Second, by laying out metadata blocks more efficiently on disk (*e.g.*, sequentially), MetaMap improves performance of metadata operations by requiring fewer disk seeks.

4.2.5 Synthetic Content Generation

Compressions services the requirements of systems oblivious to file content with data squashing and metadata remapping alone. However, many real applications and benchmarks care about file content; the DataGenerator component of Compressions is responsible for generating synthetic content to service read requests to data blocks that were previously discarded. Different systems can have different requirements for the file content and the DataGenerator has various options to choose from.

Many systems (or benchmarks) that read back previously written data do not care about the *specific* content within the files as long as there is *some* content (*e.g.*, a file-system backup utility, or the Postmark benchmark). For such systems it is sufficient to return garbage or randomly generated content in lieu of what was originally written; the simplest data generation schema in Compressions does precisely that.

Systems can read file contents and expect it to have valid syntax and semantics; the performance of these systems depend on the actual content being read (*e.g.*, a desktop search engine for a file system, or a spell-checker). For such systems, naive content generation would either crash the application or give poor benchmarking results. Compressions leverages our prior work on building the Impressions framework to generate suitable file content, using a number of existing analytical models for natural language content as discussed earlier in Chapter 3; Compressions contains a very limited in-kernel port of the file content generation capabilities of Impressions.

Finally, systems can expect to read back data exactly as they wrote earlier (*i.e.*, a read-after-write or RAW dependency) or expect a precise structure that cannot be generated arbitrarily (*e.g.*, a binary file). We have provided additional support to run more demanding applications as well; the *RAW cache* provides the necessary functionality to service applications with strict RAW requirements for file data, and the *Profile Store* supports means for applications to select or download a custom content profile specifying how Compressions should generate content suitable for the current application. We next discuss the RAW cache and the profile store in greater detail.

4.2.5.1 The RAW Cache

While Compressions is best suited for applications that do not read file data at all or require data to be read back but not necessarily match what was written, we provide support to run applications that require more precise semantics for data blocks. We term these applications as having a read-after-write (RAW) dependency for data blocks; Compressions provides a cache for storing some or all data belonging to applications that have a RAW dependency.

The RAW cache is implemented as a fixed-size circular on-disk log, the size of which can be specified by the application during initialization. Subsequently, writes to data blocks with RAW dependency are not squashed but instead written to the RAW cache; reads to these data blocks are serviced from the RAW cache instead of being synthetically generated.

The RAW cache is designed as a cooperative resource visible to the user and configurable to suit the needs of different applications. In order to decide which blocks need to be stored in the RAW cache, the application has several options to chose from with different tradeoffs for space requirement and performance:

- **All Data**: If the largest working set of the application is small enough to fit in the RAW cache in its entirety, all data blocks are written to the RAW cache. Note that this is not the same as allowing all data writes to go through to the disk (*i.e.*, no squashing) since the size of the RAW cache is fixed and data blocks do get remapped, reusing the circular log over the course of the workload execution; this is the default option in Compressions.

- **Memoization:** An optimization over the previous approach is to *memoize* or remember what previously written blocks looked liked, a concept popular in AI systems and compilers wherein the results of a set of calculations are tabulated to avoid repeating those calculations [82]. A "memoizing" DataGenerator stores the contents of previously read or written data blocks and uses them to service future requests. Memoization works best when a small set of data blocks can mimic a larger set of data blocks as used by the application to reduce the size of the RAW cache. For example, if all binary files used by an application have a similar internal structure, saving all the blocks corresponding to one file can help regenerate blocks for all other files.

- **Modified Applications:** If the application requires complete control over the file data blocks that need to be stored, it can specify so to Compressions either by providing the relevant block addresses or by placing a known *magic number* in the contents of the relevant data blocks. This approach provides efficiency in storing only the data blocks that are deemed relevant at the cost of modifying the application. For example, repeated in-house benchmarking of an application can get the most benefit out of Compressionsthrough a one-time cost of customization.

4.2.5.2 The Profile Store

Applications can require precisely structured data blocks without necessarily requiring the block being read to be exactly the same as the one written. Examples include applications working with file types that have a well defined structure that is publicly known (*e.g.*, HTML, XML, audio and video files), or ones that are specific to the application under test. In both these cases the profile store can contain a profile describing the structure of the particular file type, either selected from a built-in set of profiles, or as downloaded by the application prior to the start of benchmarking; the DataGenerator interprets the profile and generates file content accordingly.

4.3 The Storage Model

Not having access to the real storage system requires Compressions to precisely capture the behavior of the entire storage stack with all its dependencies through a model. In this section we first discuss the design goals of the model and then present the details of our disk and storage stack model.

4.3.1 Model Expectations and Goals

The usage scenario in Compressions has strongly influenced the development of our storage model; we focused on the following design goals:

- **Accuracy:** The foremost requirement for the model is to accurately predict performance for a storage device. The model should not only characterize the physical characteristics of the disk drive and other hardware components, but also the interactions of these components under different workload patterns.

- **Model overhead:** Equally important to being accurate is the requirement that the model imposes minimal overhead; since the model is in the operating system kernel and runs concurrently with workload execution, it is required to be fairly fast.

- **Portability:** The model should also be reasonably portable across different disks and storage systems. Having a generic model applicable to many different systems is hard, instead we aimed for one that required minimal manual tuning when porting across disk systems.

As a general design principle, to support low-overhead modeling without compromising accuracy, we avoided using any technique that either relies on storing empirical data to compute statistics or requires table-based approaches to predict performance [13]; the overheads for such methods are directly proportional to the amount of runtime statistics being maintained which in turn depends on the size of the disk. Instead, wherever applicable, we adopted and developed analytical approximations that did not slow the system down; our resulting models are sufficiently lean while still being fairly accurate.

The Storage Model is designed to run as a separate thread of execution concurrent with the workload execution; calls to the model are non-blocking and return immediately so as to not stall the foreground workload. The Storage Model harnesses idle CPU cycles to perform model computations in the background.

To ensure portability of our models, we have refrained from making device specific optimizations to improve accuracy. We believe our current models are fairly accurate for classes of disks and are adaptive enough to be easily configured for changes in disk drives and other parameters of the storage stack.

4.3.2 The Device Model

The device model in Compressions takes into account the following drive components and mechanisms based on a detailed model of disk drives proposed by Ruemmler and Wilkes [119]:

- Seek, rotation and transfer from disk media
- Disk caches (track prefetching, write-through and write-back)

Throughout the rest of this chapter, we will refer to the model proposed by Ruemmler and Wilkes as the RW model.

4.3.2.1 Drive Parameter Extraction

The device model requires a number of drive-specific parameters as input, for example, the disk size, rotational speed and number of cylinders; Table 4.1 contains a list of parameters modeled in the device model of Compressions. Most of the parameters are extracted from the drive itself by running a suite of carefully controlled microworkloads. When available, the drive manual serves as a useful resource both to corroborate our parameter extraction process and to get accurate values for parameters that are hard to obtain using microworkloads; we try to keep the reliance on the manual to a minimum.

Note that the above mentioned process is applicable only when the original higher capacity disk is available to the evaluator; Compressions is envisioned for use in environments when the originally desired drive itself may not be available, rather a smaller capacity drive is used as a substitute. In the latter case, it is left to the evaluator to supply the configuration of the original drive to the device model.

4.3.2.2 Modeling Seek, Rotation and Transfer

We model disk seeks, rotation time and transfer times much in the same way as proposed in the RW model. A seek in the device model is composed of a *speedup* phase, where the disk arm accelerates from the source track until it reaches half of the seek distance, a *coast* phase for long seeks, where the disk arm moves at a constant speed, a *slowdown* phase, where the arm comes to a rest near the destination disk track after deceleration, and finally a *settle* phase, where the disk controller settles the disk head onto the desired location.

As per the RW model, very short seeks (few cylinders) are modeled using only the settle time, and short seeks (few hundred seeks) are modeled primarily by the constant speedup phase. The time for short seeks is proportional to the square root of the distance between the source and destination cylinders plus the settle time. The actual parameter

Parameter	Value
Disk size	80 GB
Rotational Speed	7200 RPM
Number of cylinders	88283
Number of zones	30
Blocks per track	567 to 1170
Cylinders per zone	1444 to 1521
On-disk cache size	2870 KB
Disk cache segment size	260 KB
Disk cache number of segments	11
Disk cache read/write partition	Varies
Transfer bandwidth	133 MBps
Seek profile (long seeks; cyl $\geq$ 14000)	3800 + (cyl * 116)/1000
Seek profile (short seeks; cyl $<$ 14000)	300 + $\sqrt{(cyl * 2235)}$
Head switch time	1.4 ms
Cylinder switch time	1.6 ms

Table 4.1 **Device Model Parameters in Compressions.** *List of important parameters used to model a disk drive extracted from the drive or pre-configured by the user.*

Parameter	Value
Device driver request queue size	128 to 160 requests
Request scheduling policy	FIFO (can vary across workloads)
Delay period for timeout	3 ms

Table 4.2 **Non-device Storage Model Parameters in Compressions.** *List of important parameters used to model other parameters in the storage stack obtained through microbenchmarks or pre-configured by the user.*

values defining the above properties are specific to a drive, which we refer to as the *seek profile* of a disk drive.

For an available disk, our device model determines the actual seek profile by running a series of controlled reads to different locations on disk and measures the time taken to complete the read, varying the source and destination address pair each time in a stepwise fashion. To account for time spent in rotation, we delay the start of a subsequent read until it matches the time taken to perform one full rotation, which is subtracted out to determine the contribution of seek alone.

Rotation time is controlled by the disk rotational speed in RPM (rotations per minute); the transfer bandwidth is based on the time taken to read and write one sector from the disk media. For most cases involving random I/O, seek times dominate the rotation and transfer times; sustained transfer bandwidth becomes more relevant for sequential reads and writes.

4.3.2.3 Disk Cache Modeling

The device model also incorporates the cache resident on the disk drive. The drive cache is usually small (few hundred KB to a few MB at most in current drives) and serves to cache reads from the disk media to service future reads, or to buffer writes.

The disk drive can also "pre-fetch" disk blocks into the cache to service anticipated read requests through read-ahead from the currently read location. Disk caching can dramatically alter the timing for a disk request since the request can be serviced almost immediately as compared to an expensive seek; modeling the read cache is a crucial component of a disk model. Unfortunately, the drive cache is one of the least specified components as well; the cache management logic is low-level firmware code which is not easy to model.

To model the disk cache (a segmented cache), we found out the number and size of segments in the disk drive cache and the number of disk sector-sized slots in each segment using a set of simple microbenchmarks. By writing an increasing number of sectors to spatially separated locations, and observing when the time taken to complete the writes shows a sharp increase, we can identify the number of individual segments in the cache. By writing to spatially adjacent locations, and again increasing the number of writes till a sharp increase in time is observed, we can identify the size of an individual segment. Partitioning of the cache segments into read and write caches, if any, can be found out similarly by issuing a controlled mix of read and write operations.

We model the read cache as one with a least-recently-used eviction policy. The disk cache can also serve as a write buffer to temporarily hold incoming write requests before being written to the media (*i.e.*, a write-back cache). However, since the cache is volatile, the contents of the cache are subject to being lost on a power failure. For this reason, most system administrators prefer to turn off write buffering, or use write-through caches instead, where a write is written to the buffer but not reported complete until it is written to the media; the write buffer in that case can service future reads without reading it from the media.

To model the effects of write caching in the device model, we maintain statistics on the current size of writes pending in the cache and the time needed to flush these writes out to the media. Write buffering is simulated by periodically emptying a fraction of the contents of the write cache during idle periods in between successive foreground requests; the amount of writes flushed in any iteration are modeled in accordance with the statistics on the flush rate and the available idle time.

4.3.3 Storage Stack Model

The drive is perhaps the single most complex component to be modeled in the storage stack, but by no means is the only component that needs to be modeled. As discussed previously, the storage stack contains a number of hardware and software layers; every I/O request must flow through these layers before reaching the disk drive. Our Storage Model includes support for modeling a few other performance critical components of the stack such as the transport, and software components such as request queues. Table 4.2 contains a list of parameters modeled outside the device model in the Storage Model.

The transport is modeled rather simplistically using a constant transfer speed for the bus protocol connecting the device to the host controller. The initial "handshake" or connection setup as in SCSI is modeled using a constant time overhead. Since most drives transfer data at a slower rate than the maximum supported by the transport, the transport is rarely a performance bottleneck.

While developing the Storage Model, we found that the behavior of the request queue in the storage stack is crucial to performance, and especially tricky to model correctly. For applications that issue bursty I/O, the time spent by a request in the request queue can outweigh the time spent at the disk by several orders of magnitude.

We followed two approaches to model request queues. The first approach emulates a queue simply by maintaining the free and busy times for the disk drive as it serves a stream of request; the free and busy times are computed using the device model discussed previously. The second approach maintains a replica request queue structure, similar to the one inside a real storage stack. We discuss the two approaches in more detail next.

In the first approach, which we call the time based approach, the request queue is modeled by maintaining timing information for free and busy periods. By keeping track of when the disk will be free to service the next request in the queue, we can model the wait and service times for that request. The next free period for the disk is computed using the set of requests the disk has seen so far and computing their individual service times as modeled by the device model. Since we know when every request arrives at the disk, once we know when the disk will be free to service a particular request, we can compute the wait time and service time for that request.

We found that this approach works well for FIFO-based scheduling of I/O request, and for cases when the request stream is already optimally reordered prior to arrival at the request queue. The simple approach is also extremely efficient, requiring only a few arithmetic operations to model the wait times.

In the second approach, which we call the replica queue approach, a replica request queue structure in the Storage Model mimics the actual request queue. In this approach, the requests that arrive at the Storage Model are enqueued into the replica queue, and eventually get dispatched to the device model to obtain the disk service time. The replica request queue uses the same request scheduling policy as the original queue, and also supports "merging" of several requests into one, a common optimization found in most storage systems.

One important aspect to model a request queue accurately is potential congestion of requests. The model maintains a list of read and write requests that are waiting to be serviced; these requests are the ones that were issued subsequent to the request queue getting filled up. The list of waiting requests contains the id of the process that originally submitted a given request along with the time of the request submission. When the request queue is deemed available, the requests from the waiting list are submitted to the request queue in order of their original arrival according to the submitting process.

Both the approaches are available as part of the Storage Model and can be selected according to the needs of the experiment; the replica queue approach is more generic and models request queues more accurately than the time-based approach, while the time-based approach is relatively light weight and imposes little overhead.

4.4 Evaluation

In this section we evaluate the performance and fidelity of Compressions. We seek to answer two important questions about Compressions. First, what are the savings in terms of storage space and application runtime for benchmark workloads? Second, how accurately does the Storage Model predict the runtime and what is the overhead of storage modeling? Before proceeding with the evaluation, we first describe our experimental platform.

4.4.1 Experimental Platform

To prototype Compressions, we develop a pseudo-device driver that is inserted into the kernel; the pseudo-device driver exports itself as a "regular" disk and interposes on all traffic between the file system and the underlying hard drive. Since the driver appears as a regular device, a file system can be created and mounted on it.

We have developed Compressions for the Linux operating system with ext3 as the default file system. The hard disks that we currently model are Hitachi deskstar HDS721010KLA330 and HDS728040PLAT20 with 7200 RPM and a capacity of 1 TB and 80 GB respectively.

The Storage Model in Compressions leverages prior work on building a detailed disk simulator based on the RW model [74].

4.4.2 Performance Evaluation

We start the evaluation by answering the first question – what are the space and time savings with Compressions? The workloads that we use for this evaluation are a few microbenchmarks, `mkfs`, and PostMark. All the experiments described next were performed on the 80 GB Hitachi disk.

We first present the results of a simple experiment using the PostMark benchmark. The configuration of PostMark chosen for this experiment writes out roughly one Gigabyte of data and metadata. Figure 4.5 shows the savings in storage space required to sustain the workload; we find that the compressed file-system image provides a 10-fold reduction over the original image. Figure 4.6 shows the savings in application runtime for the same experiment; PostMark runs about 30% faster with remapping alone, and about 75% faster

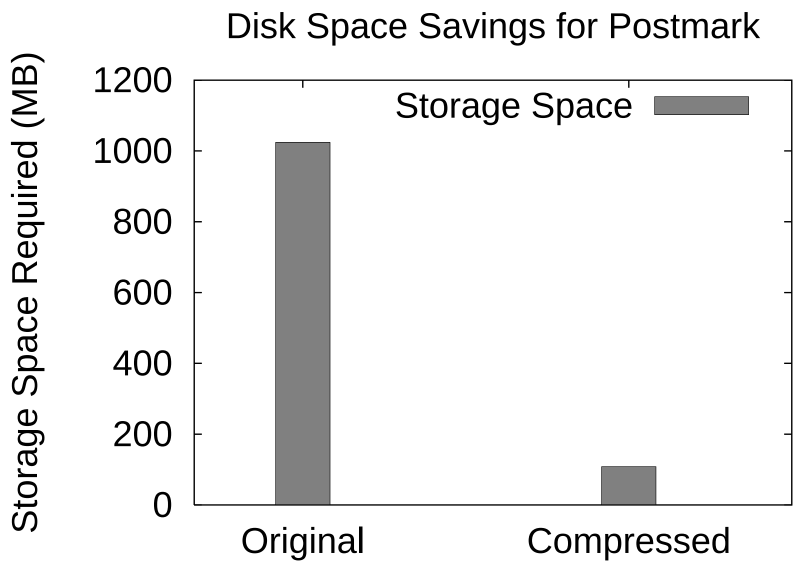


Figure 4.5 **Disk Savings for PostMark.**

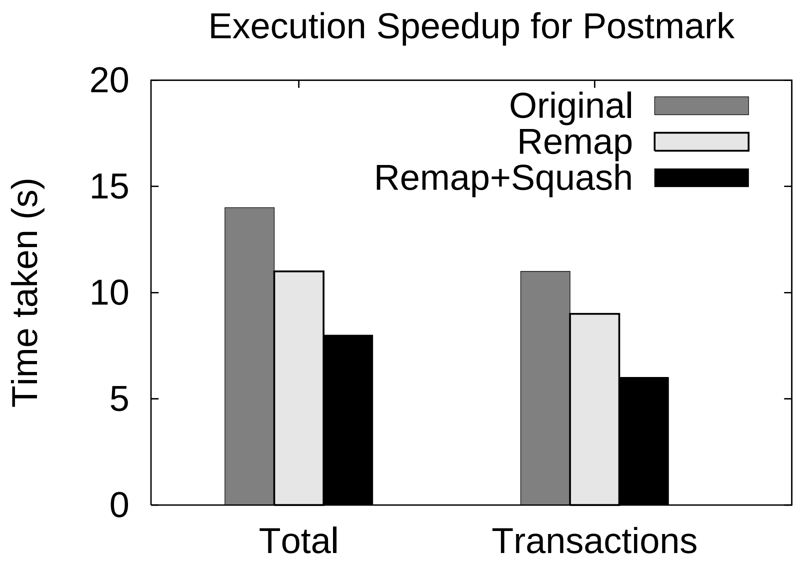


Figure 4.6 **Execution speedup for PostMark.**

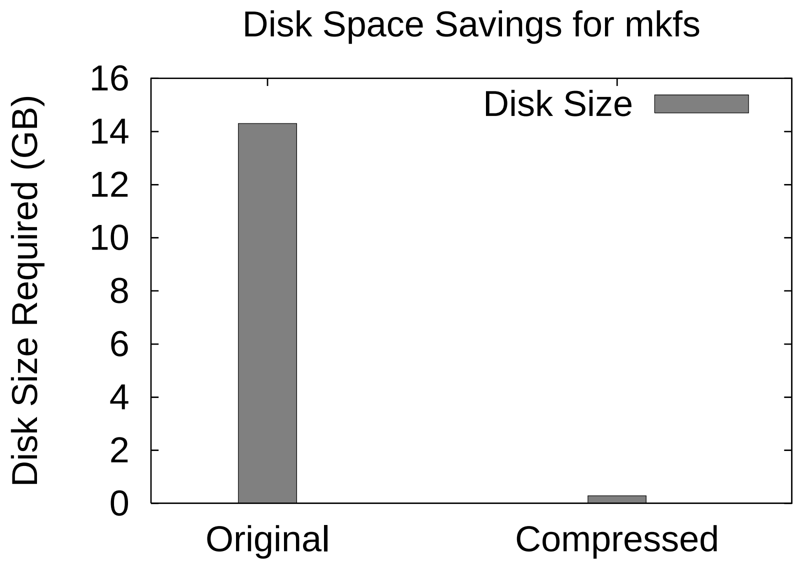


Figure 4.7 **Disk Savings for `mkfs`.**

when both remapping and data squashing are turned on. The two bars in the figure represent the total time taken by PostMark and the time spent in PostMark transactions alone.

Our second set of experiments use `mkfs` as the benchmark workload. `mkfs` is a file system utility to create a linux file system on a disk partition; we measure the time and storage space required to create Ext3 file systems on empty partitions, both with and without Compressions.

Figure 4.7 shows the size of the compressed disk partition required to sustain the creation of a 15 GB file system with Compressions providing substantial savings in storage space. Compressions exports a "fake" partition size and tricks `mkfs` into creating a file system larger than the available disk capacity.

Figure 4.8 shows the time taken to run `mkfs` with and without Compressions. The `mkfs` workload represents the worst case scenario for Compressions for two reasons. First, `mkfs` writes only metadata blocks and not any file data; applications that write to data blocks

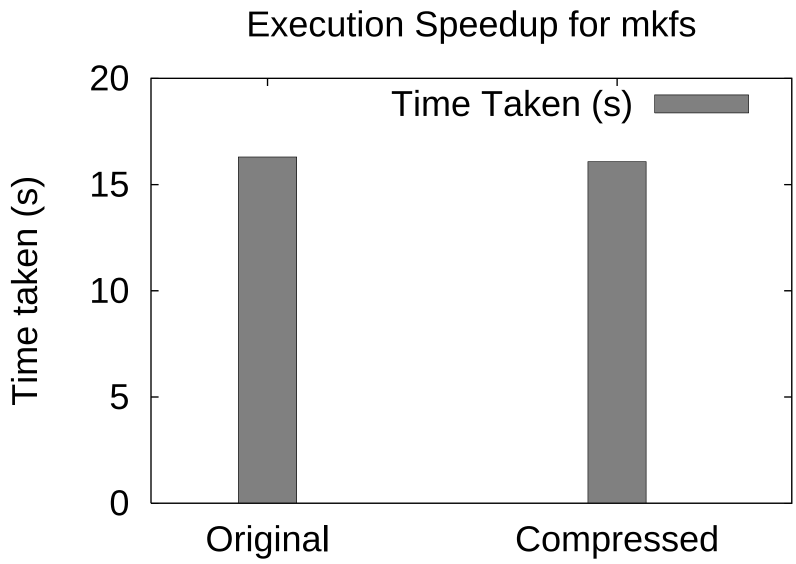


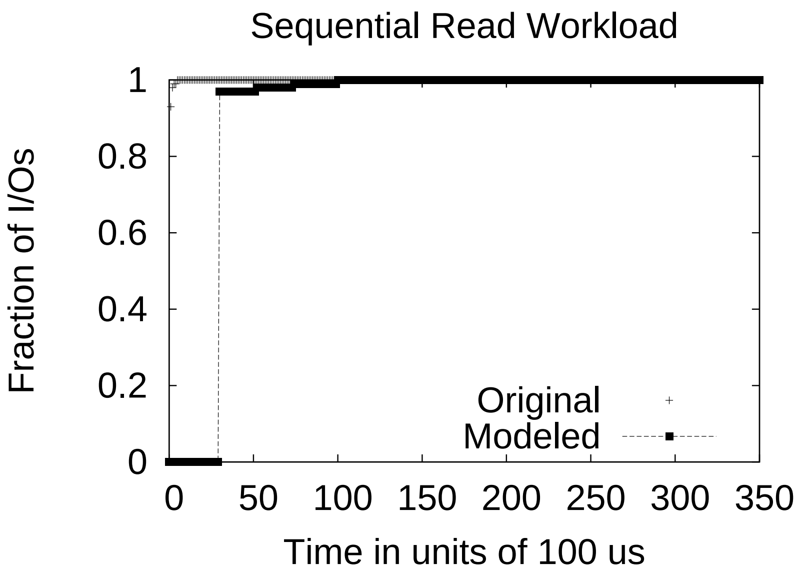
Figure 4.8 **Execution speedup for `mkfs`.**

can expect significant savings in runtime. Second, `mkfs` does not read back metadata; the sequentially laid out remapped metadata blocks thus do not provide any performance improvement for the `mkfs` workload, in spite of already having incurred the cost of block classification and remapping in Compressions.

4.4.3 Fidelity Evaluation

In the second part of the evaluation, we answer the following question – how accurately does the Storage Model predict the runtime and what is the overhead of storage modeling?

In order to provide accurate results for benchmarking, Compressions models the time taken to run a workload on the original system. We call the actual time taken by the workload to run without Compressions as the *original* or *measured* time, and the time predicted by Compressions as the *modeled* time. Figures 4.9 and 4.10 show the accuracy of modeling the runtimes by the Storage Model for four microworkloads: sequential and random



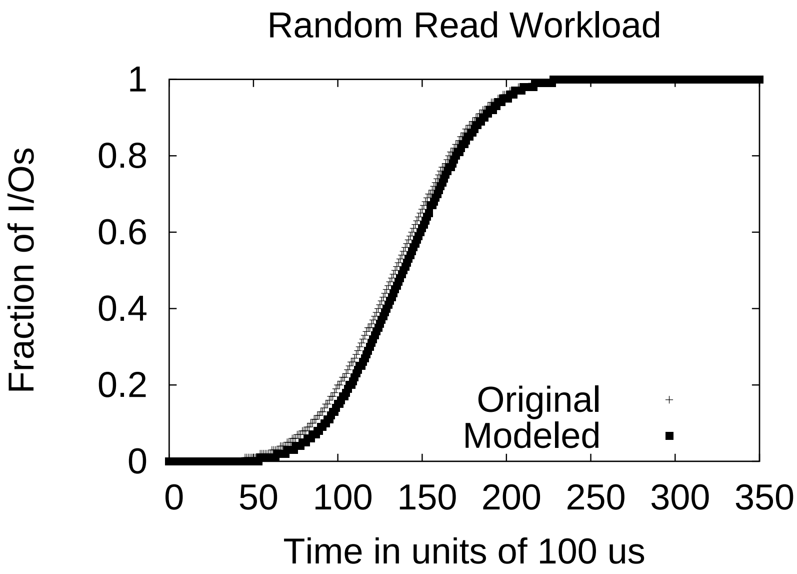


Figure 4.9 **Storage Model accuracy for Sequential and Random Reads.** *The graph shows the cumulative distribution of original and modeled times for Sequential (top) and Random (bottom) reads.*

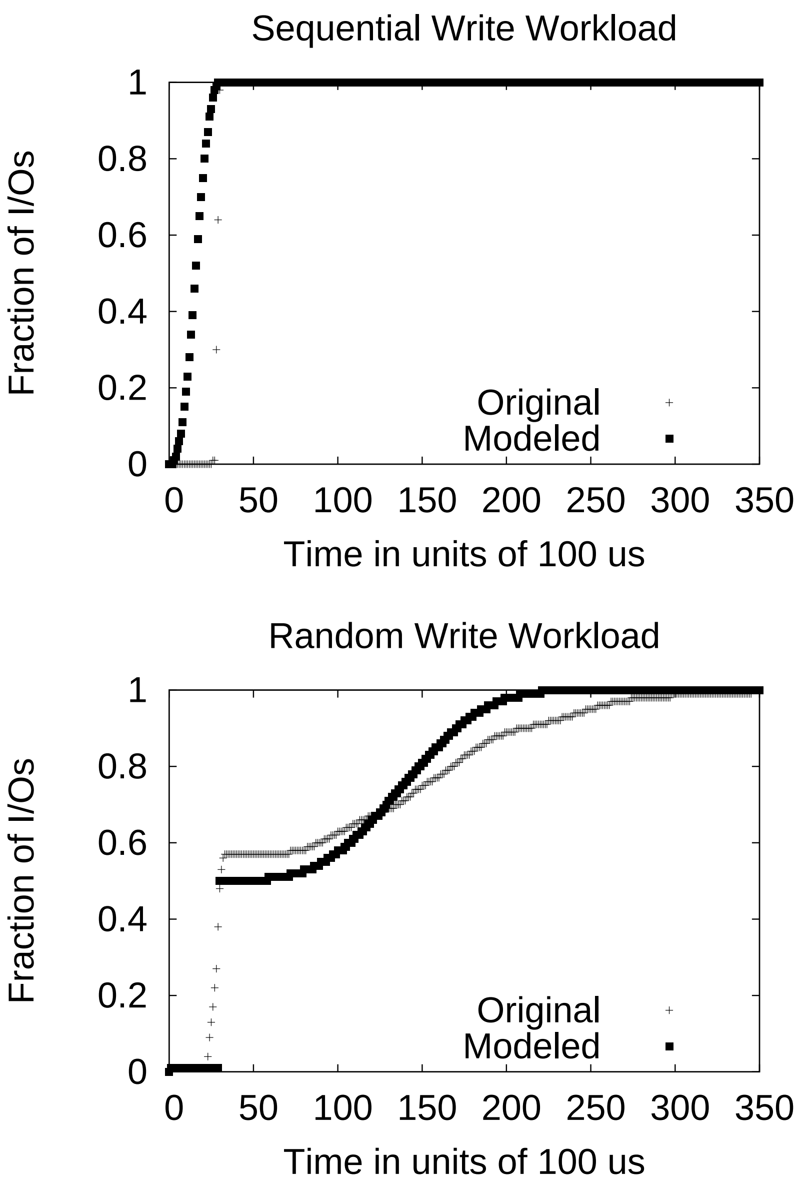


Figure 4.10 **Storage Model accuracy for Sequential and Random Writes.** *The graph shows the cumulative distribution of original and modeled times for Sequential (top) and Random (bottom) writes.*

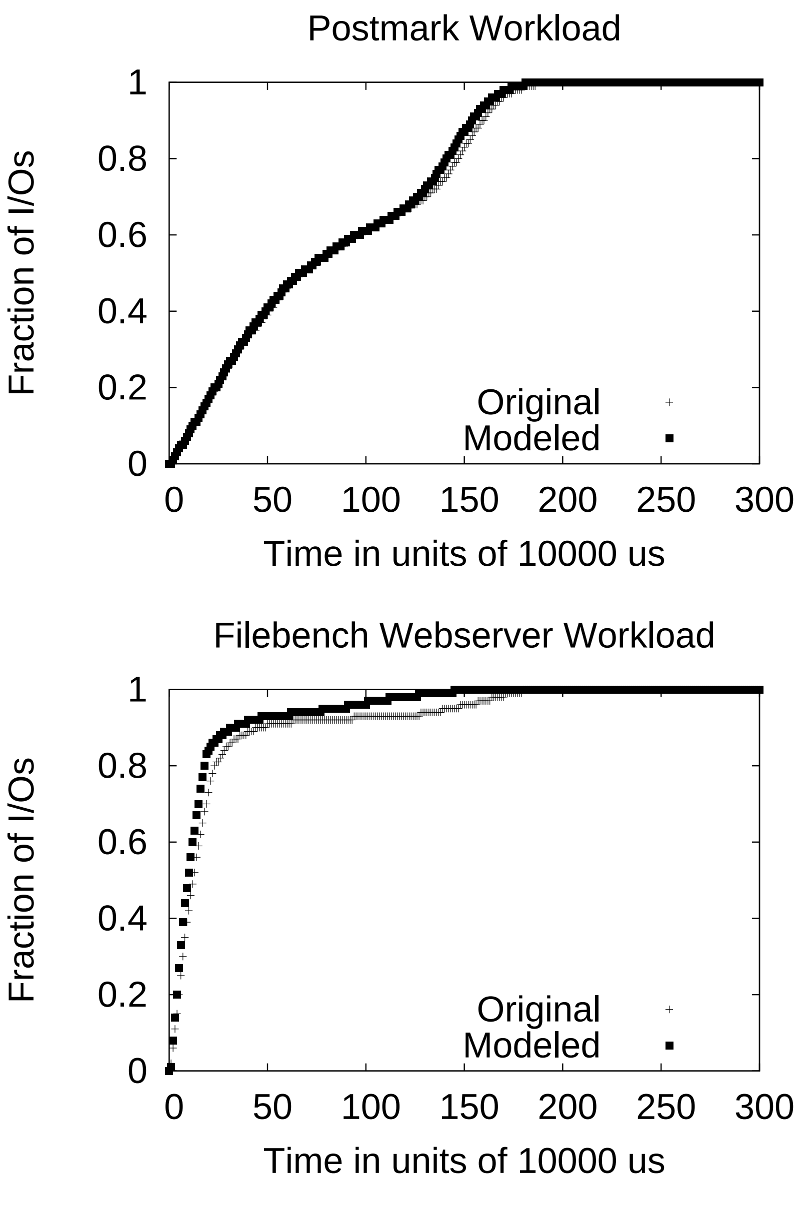
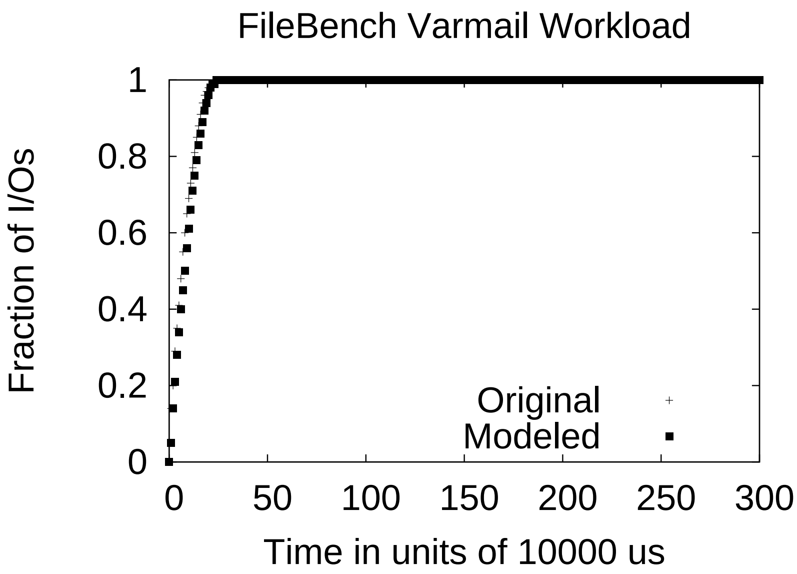


Figure 4.11 **Storage Model accuracy for Postmark and Webserver workloads.** *The graph shows the cumulative distribution of original and modeled times for Postmark (top) and Webserver (bottom) workloads.*



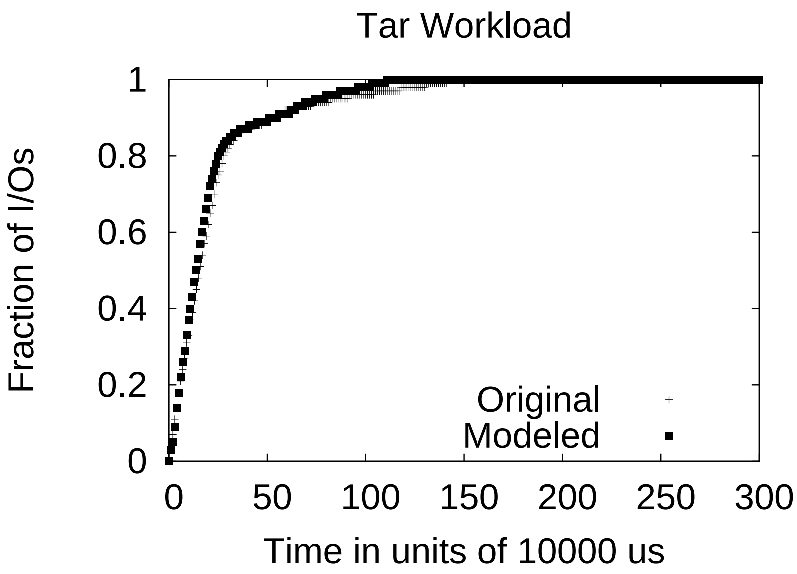


Figure 4.12 **Storage Model accuracy for Varmail and Tar workloads.** *The graph shows the cumulative distribution of original and modeled times for Varmail (top) and Tar (bottom) workloads.*

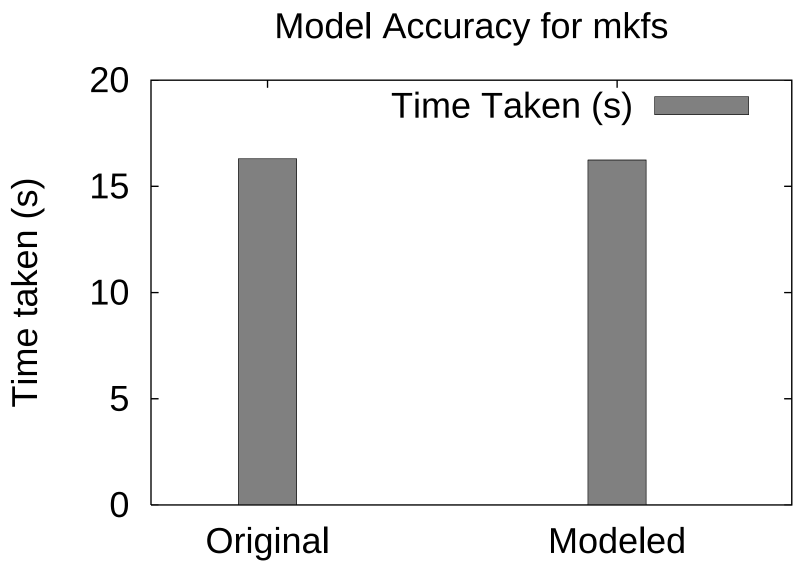


Figure 4.13 **Storage Model Accuracy for `mkfs`.**

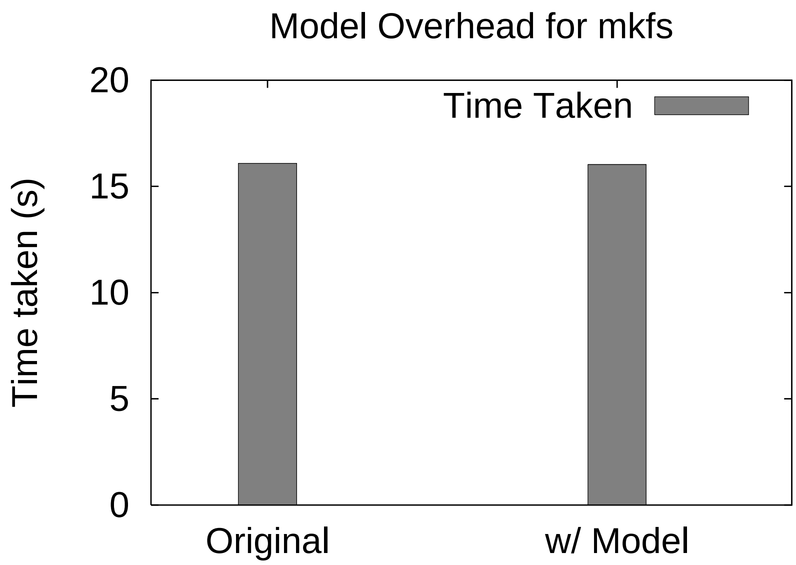


Figure 4.14 **Storage Model overhead for `mkfs`.**

Workload	Original (sec)	Modeled (sec)
Sequential Read	0.96	30.8
Random Read	133.7	139.1
Sequential Write	28.6	15.7
Random Write	74.6	75.9
Postmark	72	72
FileBench Webserver	130	130
FileBench Varmail	139	139
Tar	57	58

Table 4.3 **Accuracy of runtime modeling.** *The table shows the accuracy of modeling the total runtime of a workload. Listed here are the original measured and modeled runtimes for micro and macro workloads.*

reads, and sequential and random writes, respectively. The two lines on each graph represent the cumulative distribution of runtimes as measured for the original workload, and as modeled by Compressions. We find that the Storage Model performs quite well for simple workloads.

Figures 4.11 and 4.12 show the accuracy of modeling the runtimes by the Storage Model for four different macro workloads and application kernels: Postmark, webserver (generated using FileBench [112]), Varmail (mail server workload generated using FileBench), and a Tar workload (copy and untar of the linux kernel of size 46MB).

The FileBench Varmail workload is a NFS mail server emulation, following the workload of postmark, but multi-threaded instead. The Varmail workload consists of a set of open/read/close, open/append/close and deletes in a single directory, in a multi-threaded fashion. The FileBench webserver workload comprises of a mix of open/read/close of multiple files in a directory tree. In addition, to simulate a webserver style log, a file append operation is also issued. The configuration of the workload consists of 100 threads issuing 16 KB appends to the web-log for every 10 reads.

Overall, we find that storage modeling inside Compressions is quite accurate for all workloads used in our evaluation. The total modeled time as well as the distribution of the modeled times during workload execution are close to the observed total time and the distribution of the observed times. Table 4.3 compares the measured and modeled runtimes for the workloads described above; except for sequential reads and writes, all workload runtimes are predicted fairly accurately. The reason for the inaccuracy in our sequential workloads is a limitation of the Storage Model's disk cache modeling; we expect to resolve this issue in future versions of the Storage Model.

We now present results for `mkfs` as the workload. Figure 4.13 compares the actual runtime of `mkfs` measured without Compressions with the one predicted by the Storage Model; Figure 4.14 shows the overhead of the Storage Model itself. We find that the Storage Model of Compressions provides high fidelity modeling of benchmark workloads and imposes an almost negligible overhead.

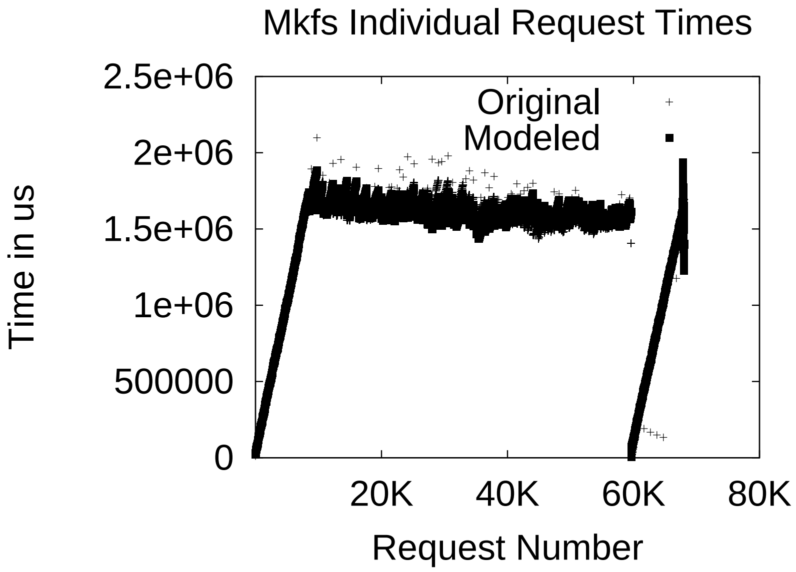


Figure 4.15 **Storage Model accuracy for `mkfs`.** *The graph shows the accuracy of modeling individual requests for `mkfs`*

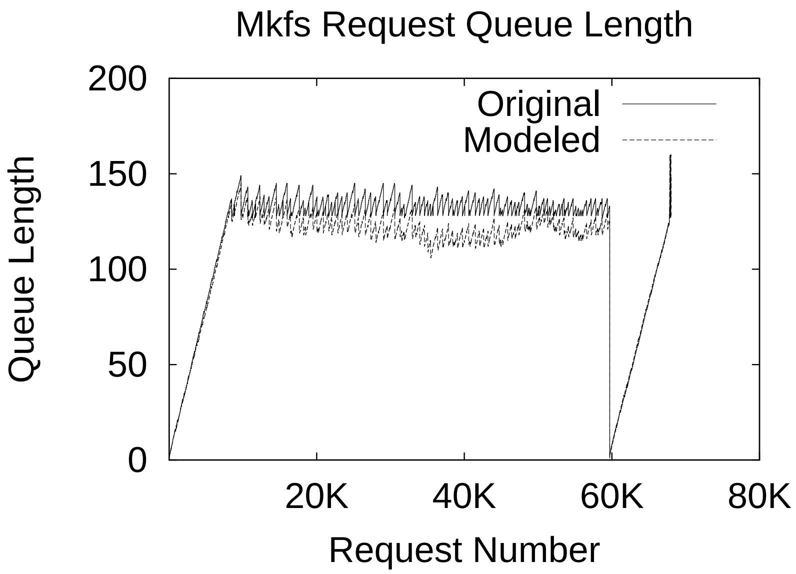


Figure 4.16 **Storage Model accuracy for modeling request queue length for `mkfs`.** *The graph shows the accuracy of modeling the amount of request queue built-up during workload execution.*

Finally, we show the accuracy of modeling individual requests for `mkfs`. We investigate the Storage Model one step further by comparing individual requests instead of aggregate times; Figure 4.15 shows the time taken by each individual request, and the corresponding modeled time. Although the goal of Compressions is to predict the total runtime of the application workload on the original storage system, each individual request is also modeled fairly consistently by the Storage Model. Figure 4.16 shows the accuracy of modeling the request queues; Compressions not only accurately models the total runtime but also the time spent in individual components of the storage stack.

4.5 Related Work

In this section we discuss related research in two areas: storage system modeling and emulation, and synthetic content generation.

4.5.1 Storage System Modeling and Emulation

The classic text on disk drive modeling by Ruemmler and Wilkes [119] describes the different components of a disk drive in detail, and evaluates the ones that are necessary to model in order to achieve a high level of accuracy. While disk drive technology and capacity have changed a lot since the paper was originally published, much of the underlying phenomena discussed then are still relevant.

Extraction of disk drive parameters has also been the subject of previous research to facilitate more accurate storage emulation. Skippy [142], developed by Talagala *et al.*, is a tool for microbenchmark-based extraction of disk characteristics. Skippy linearly increases the stride while writing to the disk to factor out rotational effects, and thereby extracts a more accurate profile for seeks. Our device model uses a similar technique but is optimized to run for large disks by introducing artificial delays between successive requests; a linear increase in stride is unacceptably slow for extracting parameters of large disks.

Worthington *et al.* describe techniques to extract disk drive parameters such as the seek profile, rotation time, and detailed information about disk layout and caching [158]. However, their techniques and the subsequent tool DIXtrac that automates the process, rely on the SCSI command interface [121], a limitation that is not acceptable since the majority of high capacity drives today use non-SCSI interfaces like IDE, ATA and SATA.

An orthogonal approach for disk modeling is to maintain runtime statistics in the form of a table, and use the information on past performance to predict the service times for future requests [13]. Popovici *et al.* develop the Disk Mimic [104], a table-based disk simulator that is embedded inside the I/O scheduler; in order to make informed scheduling decisions, the I/O scheduler performs on-line simulation of the underlying disk. One major drawback of table-based approaches is the amount of statistics that need to be maintained in order to deliver acceptable accuracy of prediction.

Memulator is a “timing-accurate” storage emulator from CMU [59]; timing-accurate emulation allows a simulated storage component to be plugged into a real system running real applications. Memulator can use the memory of either a networked machine or the local machine as the storage media of the emulated disk, enabling full system evaluation of hypothetical storage devices; Compressions can benefit from the networked emulation capabilities of the Memulator in scenarios when either the host machine has limited CPU and memory resources, or when the interference of running Compressions on the same machine competing for the same resources is unacceptable.

Similar to our emulation of scale in a storage system, Gupta *et al.* from UCSD propose a technique called *time dilation* for emulating network speeds orders of magnitude faster than available [62]. Time dilation allows one to experiment with unmodified applications running on commodity operating systems by subjecting them to much faster network speeds than actually available.

4.5.2 Synthetic Content Generation

Much in the same way as Compressions generates values for reads to invalid disk location, failure-oblivious computing uses the concept of synthetically generating values to service reads to invalid memory, while ignoring invalid writes [115]. The usage scenario is entirely different for failure-oblivious computing – enabling computer programs to remain oblivious to memory errors and continue unaffected, no attempt is made to inform the program that an error occurred upon memory access.

The importance of accurately generating synthetic test data has also been recognized in the database community. Houkjaer *et al.* develop a relational data generation tool for databases [64]. Their tool can generate realistic data for OLTP, OLAP and streaming applications using a graph model to represent the database schema, satisfying inter-table and intra-table relationships while producing synthetic row and column entries.

Aboulnaga *et al.* develop a data generator for synthetic complex-structured XML data that allows for a high level of control over the characteristics of the generated data; their tool allows the user to specify a wide range of characteristics by varying a number of input parameters [1]. Other examples of generating synthetic data include the Wisconsin benchmark [40], TPC-C and TPC-H benchmarks [148, 149], and synthetic data from the OO7 benchmark for object-oriented databases [27].

4.6 Conclusion

Motivated by our own experience (and consequent frustration) in doing large-scale, realistic benchmarking, we have developed Compressions, a benchmarking system that allows one to run large, complex workloads using relatively smaller storage capacities. Compressions makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system by transparently scaling down the storage capacity required to run the workload. The required disk size under Compressions can be orders of magnitude smaller than the original while also allowing the benchmark to execute much faster. Compressions ensures the accuracy of benchmarking results by using a model of the disk and the storage system to compute the runtime of an application on the original unmodified system.

Chapter 5

Generating File-System Benchmark Workloads

Another crucial requirement for benchmarking apart from the benchmark state is the benchmark workload, without which no benchmarking can proceed. Like benchmarking state, the benchmark workload should also be representative of the target usage scenario, in this case the real-world applications running on the system. While creating a benchmark workload, care must be taken to ensure that the workload is easy to reproduce so as to enable comparison across systems.

To evaluate the performance of a file and storage system, developers have a few different options, each with its own set of advantages and disadvantages.

- **Real Applications:** One option for evaluating a file or storage system is to directly measure its performance when running real I/O-intensive applications. The obvious advantage of benchmarking with real applications is that the performance results can correspond to actual scenarios in which the system will be used and that users care about. However, the problem is that real I/O-intensive applications can be difficult to obtain, to setup, and to configure correctly [144].

Some systems have used production workloads to evaluate performance, mostly ones developed in large corporate environments where production workloads are available in-house. Examples of such systems include the Google File System [51], IBM GPFS [122] and NetApp Data ONTAP GX [45]. Often system evaluators compromise by running "real applications" that they are the most familiar with, such as compiling an operating system kernel or untarring a source tree. While these workloads are easier to setup, they may not be fully representative of the end applications.

- **Microbenchmarks of Application Kernels:** A second option is to run application kernels instead of the full applications themselves. For example, instead of configuring and stressing a mail server, one can instead run the PostMark [70] benchmark, which attempts to produce the same file system traffic as a real mail server. Other examples of kernels include the original [65] and modified Andrew Benchmarks [99], SPC-1,2 [138], and the TPC Suite [147, 148]. The disadvantage of using kernels is the loss of representativeness. While these kernels are simpler to run, they have the fundamental problem that their simplifications both may make them no longer representative of the original workload and enable system designers to artificially optimize to specific kernels.

- **Trace replay:** A third option is to replay file system traces that have been previously gathered at various research and industrial sites. Examples of traces include file system traces from HP Labs [114] and a collection of I/O and file system traces available through SNIA's IOTTA Repository [133]. Replaying file system traces eliminates the need to setup and recreate the original applications. Trace replay however has challenges of its own.

In particular, replaying the trace while accurately preserving the original timing [14] and accounting for dependencies across I/O requests [85] are non-trivial problems. In addition, traces are often large, unwieldy, and difficult to use.

- **Synthetic workloads:** A final option is to run synthetic benchmark workloads that are designed to stress file systems appropriately, even as technologies change. Synthetic workloads, such as IOZone [98], SPECsfs97 [157], SynRGen [44], fstress [12], and Chen's self-scaling benchmark [31], contain a mix of POSIX file operations that can be relatively scaled to stress different aspects of the system.

The major advantages of synthetic applications is how simple they are to run and that they can be adapted as desired. However, the major drawback of synthetic workloads is that they may not be representative of any real workloads that users care about.

In practice, we find that evaluators often use a combination of different types of workloads and that there is significant diversity in the choices. A detailed survey of file system benchmarks used in publications from top systems conferences such as SOSP, OSDI and FAST was conducted by Traeger *et al.* [145]; the survey contains details on benchmark usage, including descriptions of the workload characteristics and their configurations. Here we summarize some of the important findings from the survey.

On the whole, synthetic benchmark workloads are much more popular than using real applications or trace replay. We suspect that this can be attributed to the ease of use with which synthetic workloads can be employed. The most popular individual benchmark workload according to the survey is PostMark [70], written at Network Appliance. PostMark strives to measure performance for email, netnews and web-based commerce applications. Other popular benchmarks include the Andrew benchmark and its variants, first developed in 1988 at CMU; and the Cello disk traces [118] collected at HP Labs in early 90's. Benchmarks from the TPC suite [147, 148] released in 1990 and 1992, and specSFS [157] from 1993 are also used often.

Any synthetic benchmark workload referred to in the survey has one or more of the following drawbacks: the benchmark workload can be obsolete and not representative of any real application the user cares about, the benchmark workload is not easily reproducible, the parameters for various workload properties can be incorrect or outdated, and finally, the default or primary configuration of the benchmark workload can be unsuitable or misleading.

As noted earlier, most of the popular benchmark workloads appearing in this survey were written over a decade ago, for different operating environments. Usage and access patterns evolve over time and also vary with different environments [77, 116, 153]; improvements need to be made to these workloads to reflect the changes. In general, it is hard to maintain the authenticity of a benchmark workload once it is deployed. A periodic overhaul of the workload is needed to prevent it from getting outdated; no automated means exist to keep a given synthetic workload in-sync with its real counterpart.

Benchmark workloads can also be hard to reproduce, much like the difficulty in generating reproducible file-system images for benchmarking. Trace replays and production applications used as benchmark workloads are especially hard to reproduce; they require preserving the original timing information and accounting for dependencies across I/O requests. Ill-defined and incompletely specified workloads are also not accurately reproducible.

Additionally, distributions for commonly used file system attributes like file sizes have changed over time as well; most benchmark workloads have not kept pace with recent trends. One example is the Postmark benchmark workload, a single threaded application that works on many short-lived, relatively small files. The workload consists of several *transactions* where each transaction is one of a file create or delete; or a file read or append operation. The benchmark begins by creating a random pool of text files with uniform distribution of sizes between high and low bounds. The file create operation creates a new file and writes random text to it. File delete operation deletes a randomly chosen file. File read reads a random file in its entirety and a file write appends a random amount of data to a randomly chosen file. Distribution of email file sizes has changed over the years. One indicator of this trend is the size of Personal Storage Table (or `.pst`) files in Microsoft Windows used to store Outlook messages locally, as observed in our metadata study (Chapter 2); PostMark has not kept pace with this trend.

Finally, a poor workload configuration is often the cause for misleading results. Given a choice of workload, it is left to the evaluator to choose a configuration suitable for the benchmarking experiment. The configuration can control various parameters of the workload such as its working set size, number of I/O operations or transactions, options for I/O modes (*e.g.*, buffered and non-buffered), and number of concurrent threads. A poor configuration choice can defeat the purpose of benchmarking even for a workload that is perfectly reasonable otherwise. Synthetic workloads and application kernels are especially prone to misconfiguration. A continuing example of such a workload is Postmark. Through our own evaluation and as noted by others [125, 145], we find that PostMark has several shortcomings. One glaring example is the default configuration of PostMark. Not only it is obsolete, the amount of I/O traffic generated is not nearly enough to stress modern computer and I/O systems, the entire working set easily fitting in memory for any modern machine.

Given the popularity of synthetic benchmarks and their consequent misuse in staking claims about performance, we believe that the ideal benchmark for file and storage systems combines the *ease of use* of synthetic benchmarks with the *representativeness* of real workloads. While we have not perfected a complete system capable of creating realistic benchmark workloads, we have made a promising start. The goal of this chapter is to describe how one might create realistic synthetic benchmarks using our proposed technique.

Specifically, our approach is to provide a tool that enables one to create a synthetic benchmark that is functionally equivalent to a given real application or a workload comprising of a mix of applications; that is, the synthetic benchmark stresses the underlying system in the same way as the original set of applications.

In this chapter, we first describe our initial steps in this direction by building a tool, CodeMRI (an "MRI" for Code, if you will) in Section 5.1, and second, in Section 5.2, we discuss future work in designing automated workload generators using CodeMRI as a building block. To bring the chapter to an end, we discuss related work in Section 5.3.

5.1 Generating Realistic Synthetic Benchmarks with CodeMRI

Determining whether or not two workloads stress a system in the same way is a challenging question; certainly, the domain of the system under test has a large impact on which features of the two workloads must be identical for the resulting performance to be identical. For example, if the system under test is a hardware cache, then the two workloads might need to have identical addresses for all issued instructions and referenced data; on the other hand, if the system under test is a network protocol, the two workloads might need to have the same timing between requests to/from the same remote nodes. Therefore, the specific features of the real workload that must be captured by the synthetic benchmark depend on the system. For file and storage systems, one might believe that an equivalent synthetic workload could be created by simply mimicking the system calls through the file system API (*e.g.*, read, write, open, close, delete, mkdir, rmdir). Given that tools such as strace [140] already exist to collect system call traces, creating such a synthetic workload would be relatively straight-forward. The problem is that system calls that appear identical (*i.e.*, have the exact same parameters) can end up exercising the file system in very different ways and having radically different performance.

File systems are complex pieces of system code containing hundreds of thousands of lines of code spread across many modules and source files. Modern file systems contain code to perform caching, prefetching, journaling, storage allocation, and even failure handling; predicting which of these features will be employed by a given system call is not straight-forward. Furthermore, the storage devices that are physically storing the data have complex performance characteristics; accesses to sequential blocks have orders of magnitude better performance than accesses to random blocks.

Consider the example of a `read` operation issued through the API. This read might be serviced from the file-system buffer cache, it might be part of a sequential stream to the disk or a random stream, or could involve reading additional file system meta-data from the disk. Similarly, a `write` operation might allocate new space, overwrite existing data, update file-system metadata, or be buffered as part of a "delayed write". In each of these cases, the exercised code and the resulting performance will be significantly different.

Our hypothesis is that to create an equivalent synthetic benchmark for file and storage systems, one must mimic not the system calls, but the *function calls* exercised in the file system during workload execution, in order to be functionally equivalent. We believe that if two workloads execute roughly the same set of function calls within the file system, that they will be roughly equivalent to one another.

CodeMRI uses detailed analysis of the source code for the file system under test to understand how a workload is stressing it. Specifically, CodeMRI measures function-call invocation patterns and counts to identify internal system behavior. Our initial results in applying CodeMRI to macro-workloads and benchmarks such as PostMark [70] on the Linux ext3 [150] file system are promising.

First, CodeMRI is able to deconstruct complex workloads into micro-workloads; each micro-workload contains system calls (*e.g.*, read and write) with known internal behavior (*e.g.*, hitting in the file buffer cache or causing sequential versus random disk accesses). In other words, with good accuracy, we are able to identify that a "real" workload, such as PostMark, performs the same set of file system function calls as a combination of system calls with certain parameters. Second, to a limited extent, we are able to predict the runtime of the workload based on this set of constituent micro-workloads.

5.1.1 Experimental Platform

CodeMRI was developed and tested on a dual core Intel Pentium III-Xeon machine with 512K Cache and 1GB of main memory with Ext3 as the test file system on the Linux operating system. The hard drive was an IBM 9LZX SCSI disk with a rotational speed of 10000 RPM, on-disk cache of 4MB, and a capacity of 9GB.

CodeMRI requires tracing function invocations during workload execution, we used a statically instrumented version of the linux kernel compiled using the `gcc -finstrument-functions` option. The instrumentation inserts *hooks* to special profiling functions for each entry and exit point of a function in the file system. The hook functions are called with the addresses of the caller and callee functions, a timestamp, and any additional information that needs to be captured at runtime.

5.1.2 Micro-workloads and Microprofiles

The goal of CodeMRI is to be able to construct synthetic equivalents of real workloads, but there are two challenges in solving this problem. First, we need to accurately deconstruct real workloads into constituent *micro-workloads*. A micro-workload is a simple, easy to understand workload such as the `read` system call in its many forms, each with the same behavior (cached or not, sequential or random). Second, we need to be able to use the set of micro-workloads to compose a synthetic equivalent of the original workload.

We plan to approach this problem by leveraging two sources of information. First, we leverage domain knowledge about the system under test. Second, we use tracing to obtain useful information about the workload execution and the system under test.

Domain knowledge about file systems consists of basic knowledge about the different features that it provides, such as caching and prefetching. This is useful to know because different workloads can exercise different system features that CodeMRI needs to identify. The domain knowledge guides the tracing of execution profiles for micro-workloads. For example, we need to have an execution profile for a *cached read*. The execution profile is simply a list of unique functions and their invocation counts during a particular workload execution.

For tracing the workload execution, we believe function invocation patterns and invocation counts provide the amount of detail necessary to understand the benchmark workload and the functionality that it exercises. This constitutes the *execution profile* of the workload.

In order to address the first challenge – to breakdown real workloads into simpler micro-workloads, we compare the execution profile of a real workload with the set of execution profiles of individual micro-workloads. We call the execution profile of a micro-workload a *microprofile*. To address the second challenge – to synthesize a synthetic equivalent, we intend to compose the microprofiles together, along with timing and ordering information. Microprofiles are thus the building blocks for achieving both our objectives.

5.1.3 Building Microprofiles

The first step in building CodeMRI is to identify a comprehensive set of micro-workloads and build their microprofiles. We achieve this by running all the system calls through the file system API, under the effect of various file-system features. For example, in the case of a `read` system call, we identify two features that matter: whether the read is cached or not, and whether it is random or sequential; we then look at all four combinations of the workload as a function of the request size and build microprofiles for uncached read, cached read, sequential read, and random read.

Through our experiments we find that keeping track of sets of function invocations and their counts, during a workload execution, allows us to build accurate microprofiles. We also observe that it is cumbersome and unnecessary to keep the entire execution profile – instead we select a small set of function invocations that uniquely characterize a microprofile. We call this the *predictor set* of a microprofile, and consequently the corresponding micro-workload. A predictor set typically consists of one to few tens of function calls, depending on the number of micro-workloads. The intuition behind this approach is that each function contributes towards completion of a higher level workload such as a `read`.

Each function thus serves as the smallest unit of "useful work". The goal is to identify a set of functions that uniquely represent the higher level workload.

For example, in the absence of any other workload, a `read` system call can be identified simply by observing calls to the `sys_read` function. However, if we need to distinguish between sequential and random reads, then `sys_read` will be shared by both and useless to distinguish; in that case, an extra function call `ext3_readpages` helps us differentiate the two. The reason behind that is sequential reads trigger block prefetching and a fraction of blocks do not need to be explicitly requested for from the disk thereby altering the invocation count of the function for the two workload patterns.

The larger the number of micro-workloads, the greater the number of predictor sets needed to differentiate amongst them; consequently, the size of the predictor set is directly proportional to the number of micro-workloads that need to be separately accounted for. In order to identify the set of function calls that constitute the predictor set for a workload from amongst all the possible functions that contribute, we define some metrics to help automate the task.

We have three quantitative metrics associated with a predictor set – *slope*, *uniqueness* and *stability*. Two of these, uniqueness and stability (both on a scale of 0 to 1), are used in the selection of predictor sets. Each member function in a predictor set has a *slope* which characterizes the rate of change of invocation count with change in some workload parameter (such as request size). We define the *uniqueness* of a predictor set towards a micro-workload (such as `read`) as its affinity with the micro-workload. A uniqueness of 1 implies that the particular predictor set is invoked exclusively during this workload's execution, while 0.5 implies that it has an equal affinity with another workload, and 0 makes it irrelevant for that workload. The *stability* of a predictor set is a measure of the variability of function-invocation counts as some workload parameter is varied. A perfectly stable predictor set (i.e., with stability equal to 1) will scale proportional to the *slope*, as

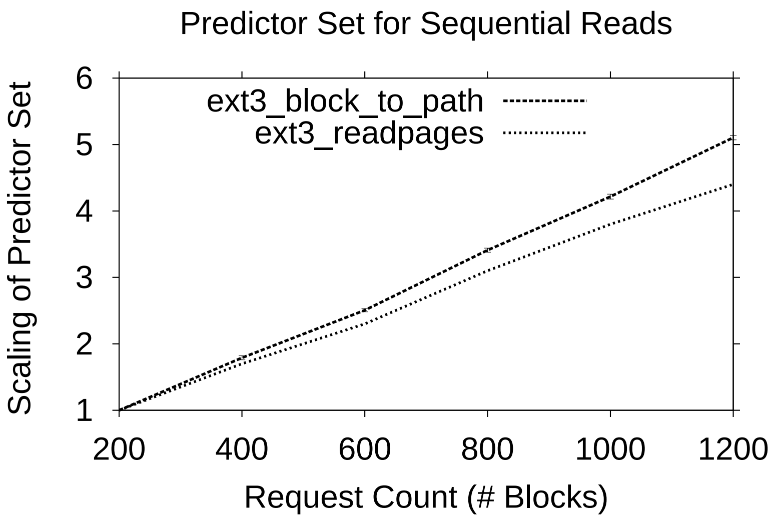


Figure 5.1 **Predictor Set for Sequential Reads.** *Having two member functions with scaling slopes equal to* 0.85 *and* 0.895*,* uniqueness *of* 1*, and* stability *very close to* 1

the request size of the workload is increased, for instance. A stability of 0 means that the predictor set scales in a completely uncorrelated fashion and is useless for prediction. Thus, an ideal predictor set for a given workload is one having both uniqueness and stability equal to 1.

Figure 5.1 shows a simple example of the predictor set for sequential reads having two member functions with scaling *slopes* equal to 0.85 and 0.895, *uniqueness* of 1, and *stability* very close to 1. This makes it a good candidate for being a predictor set to identify sequential reads.

To compute the slope for member functions in a predictor set, we keep track of their invocation counts, as we vary a workload parameter. This is done for a small "training range" to create a model. For example, in Figure 5.1, the training range for the request size model is from 200 to 1200 file system blocks. Figure 5.2 repeats the experiment for random reads. Notice that the slope of the invocation count for random reads is different as

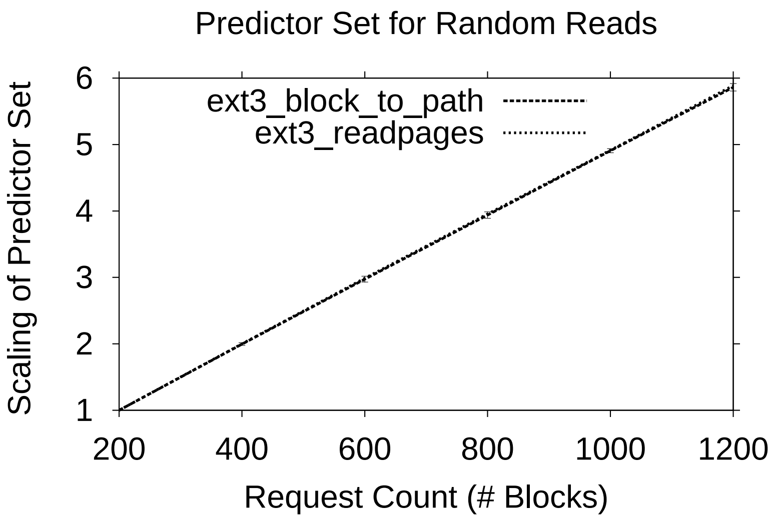


Figure 5.2 **Predictor Set for Random Reads.** *Having two member functions with scaling slopes equal to* 0.98 *and* 0.97*,* uniqueness *of* 1*, and* stability *very close to* 1

compared to sequential, and is used to distinguish the two. Figure 5.3 shows accuracy of prediction for writes.

In our (limited) experimental evaluation, we found that the predictor sets identified by CodeMRI are stable beyond the training-model range. In practice, stability can be affected by "cliffs" for different regimes of workload execution where linear interpolation will be insufficient.

The predictor set allows us to accurately predict the extent of the corresponding micro-workload. For example, if we observe a call to the function `ext3_readpages` and `ext3_block_to_path` a certain number of times, we can infer the corresponding bytes of random read being performed. Similarly the predictor set for cached reads will correspond, as shown in Figure 5.4, to the amount of bytes being serviced from the buffer cache during a `read` operation.

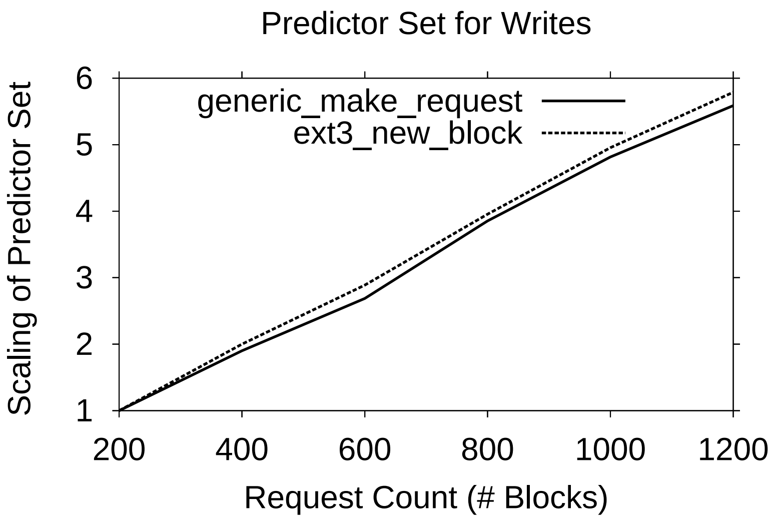


Figure 5.3 **Predictor Set for Writes.** *Having two member functions with scaling slopes equal to* 0.97 *and* 0.93*,* uniqueness *of* 1*, and* stability *very close to* 1

In Figure 5.5 we show few function calls that are not good predictors for the above example, *i.e.*, distinguishing amongst sequential and random reads. These functions are either not invoked at all during a read call, or are invoked but their invocation count does not change predictably with the amount of data being read, rendering them useless.

The choice of a predictor set for any workload is not constant. For a single micro-workload, it is easy to find a predictor set with uniqueness equal to 1. However, for a real, complex workload, consisting of potentially tens to hundreds of micro-workloads, there can be significant overlap in the set of function invocations amongst the different micro-workloads, such that finding a predictor set for each of them is not straightforward. The size of the predictor sets depends on the number of micro-workloads to be deconstructed from the real workload. The more complex the real workload, the greater the number of functions required to construct predictor sets for each of the micro-workload.

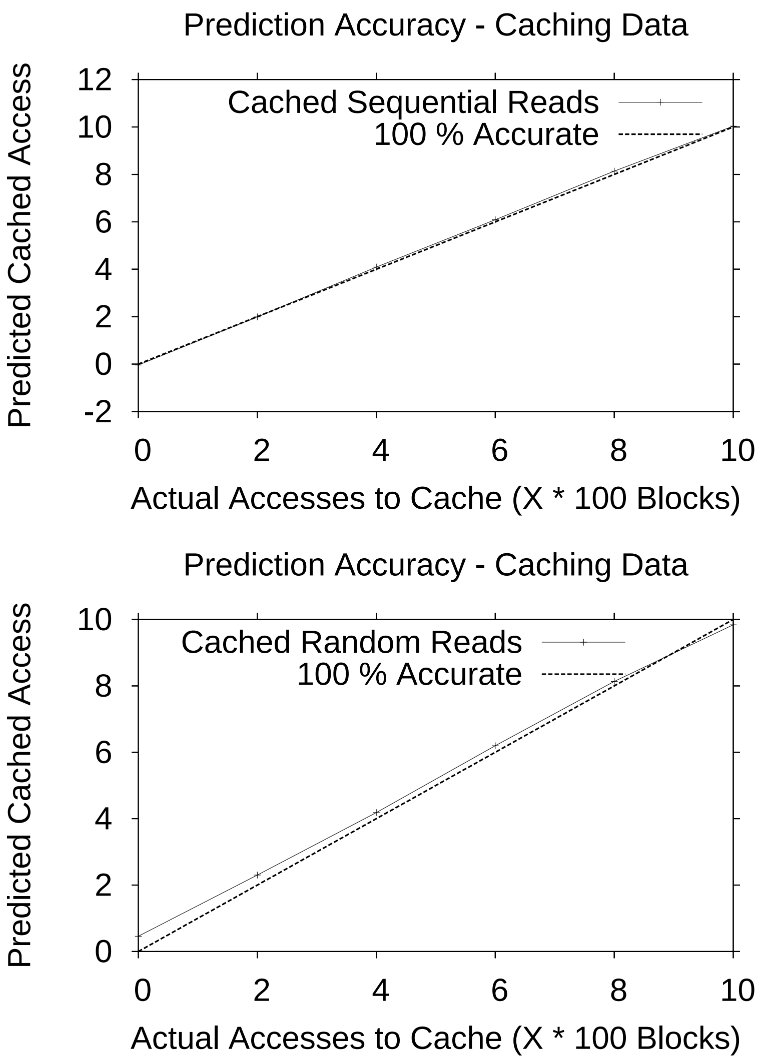


Figure 5.4 **Accuracy for Cached Reads.** *Figure shows the predicted and actual reads serviced from the cache during sequential and random read workloads.*

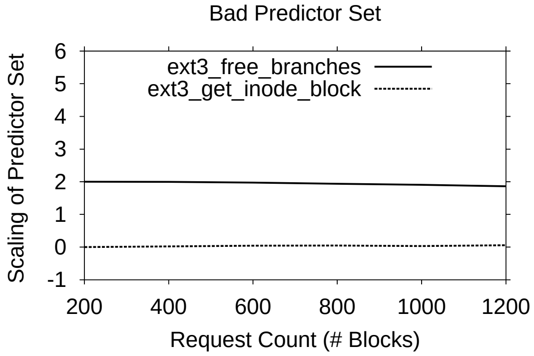


Figure 5.5 **Example of a Bad Predictor Set for Reads.** *Invocation of function ext3_free_branches does not change with request and ext3_get_inode_block does not get invoked very much at all. Slopes for both predictors equals to* 0*,* uniqueness *of* 0*, and* stability *very close to* 1

CodeMRI consists of an algorithm based on linear-programming (LP) to select predictor sets, attempting to maximize the uniqueness for each of the micro-workloads. The LP problem constraints are in the form of minimum acceptable values for slope and stability, wherein the predictor set consists of the top-K functions that satisfy the given criteria, with K being the number of micro-workloads under consideration. In the absence of any function that satisfies the given slope and stability criteria, the conditions are relaxed until a match is found.

During our experiments with the micro-workloads and in understanding the structure of the file system source code, we find that in practice, micro-workloads exhibit a natural division of function invocations, making it feasible to select predictor sets even for a mix of applications; if each workload is run in isolation it is relatively straightforward to select a good quality predictor, but as more workloads get added to the mix, finding predictors becomes harder. We have not tested CodeMRI for complex workloads and it is likely that our techniques will need refinement for use in such scenarios.

Micro Workload	Variations	Parameter Extent or Count
Read	sequential or random cached or not cached	degree of randomness degree of caching
Write	sequential or random	degree of randomness
POSIX calls	open, mkdir, rmdir, create, delete, close cached calls e.g., open after create	count degree of caching

Table 5.1 **List of Micro-Workloads Tested.** *The table lists the various micro-workloads and their variations that were deconstructed with CodeMRI, along with the parameter of interest that was successfully predicted.*

5.1.4 Using Microprofiles for Deconstruction

We now describe the use of microprofiles to deconstruct workloads. We present our discussion with increasing complexity of benchmark workloads:

- micro-workload, such as a `read` or `write`
- macro-workload consisting of micro-workloads
- macro-workload under caching
- application kernel: PostMark

Table 5.1 shows the list of micro-workloads that we have experimented with and are able to predict with good accuracy.

The accuracy continues to be good for macro-workloads. Figure 5.6 shows the accuracy of prediction for a macro-workload consisting of `reads`, `writes` and system calls such as `open` and `close`, as we vary the request size. The leftmost graph shows the accurate prediction of random writes, and the middle one shows the ratio of writes to sequential and random reads as predicted. The rightmost graph in Figure 5.6 shows the accuracy of prediction of reads under caching. In these graphs, the "Model" line represents the training range on which the slope for prediction was computed.

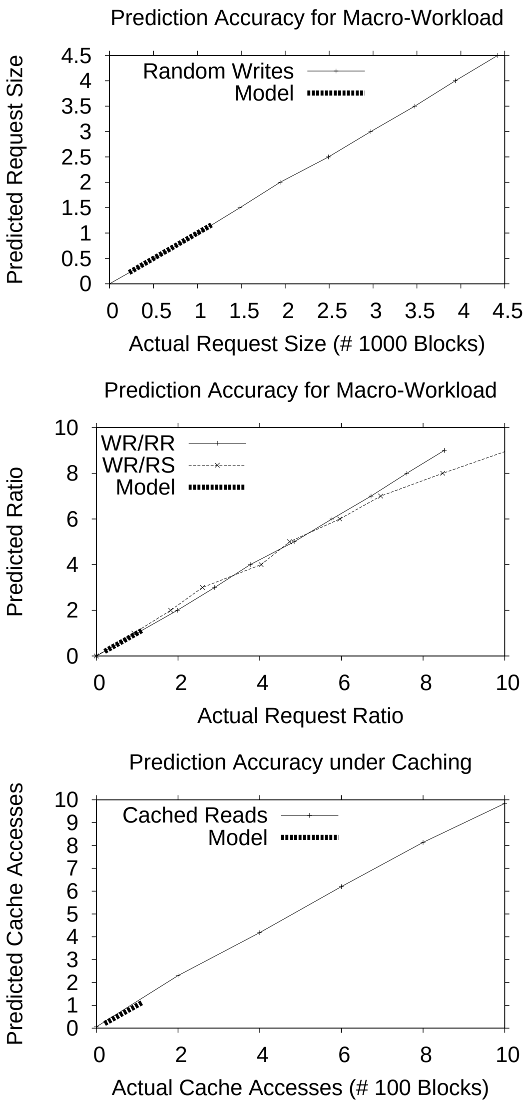


Figure 5.6 **Accuracy for Macro-Workloads.** *The leftmost graph shows the accurate prediction of random writes in a macro-workload consisting of writes and reads (both random and sequential). The middle graph shows the ratio of random writes (WR) to read sequential (RS) and random (RR) as predicted by CodeMRI. These ratios show that the relative counts across workloads are also accurate. The rightmost graph shows the shows the accuracy of prediction under caching. The Model line represents the training range on which the slope was computed.*

CodeMRI is thus able to accurately identify workloads well beyond the small training range of request sizes for which the slope model was computed. Only for larger deviations from this range do we observe inaccuracy.

In order to verify whether this deconstructed workload has any correlation with actual performance, we use it to predict performance and compare with the actual measured performance. The hypothesis is that if the deconstruction is accurate, then the sum of time taken by the individual micro-workloads should be close to the actual measured time. To predict performance once we have identified the set of micro-workloads, we simply add the time it takes to run them individually. This is a coarse estimate, as it does not take into account dependencies amongst the micro-workloads.

Figure 5.7 shows an example of this for a macro-workload consisting of random and sequential reads, `mkdir`, `create`, and `delete` operations. The left graph highlights that the primary contributor(s) to performance can be different from the expected ones, and CodeMRI can identify the real contributors. The "issued operations" are the ones issued through the file system API. The "actual operations" are the ones being actually issued by the file system to the disk, and not serviced from cache. The "predicted operations" are the ones identified through CodeMRI.

In order to predict the runtime, we need to know the time it takes to run the individual micro-workloads (or the predicted operations in this case); this timing information is collected during the initial fingerprinting phase of every micro-workload as it is run with varying request sizes. To compute the predicted runtime, we simply add the individual runtimes of all micro-workloads.

In this example, random reads contribute much less to overall runtime than sequential reads and mkdir. The stacked bar graph on the right shows the predicted runtime contributions from individual micro-workloads. We see that the predicted cumulative runtime matches closely with the measured runtime, demonstrating the accuracy of CodeMRI.

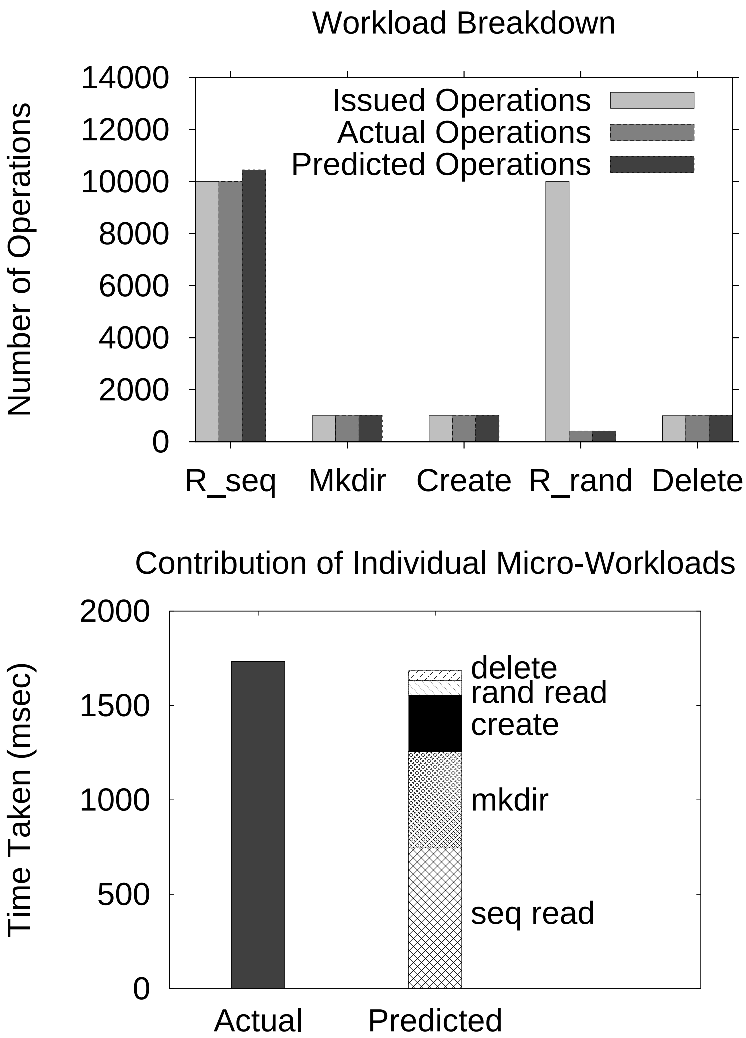


Figure 5.7 **Macro-workload Deconstruction.** *The macro-benchmark consists of* `mkdir`*,* `create`*,* `delete`*, repeated random reads to a small file, and sequential reads to a large file, resulting in random reads hitting the cache. The graph on the left shows the deconstruction of this macro-workload by CMRI which identifies the effective (reduced) count of random reads. The "issued operations" are the ones issued through the file system API. The "actual operations" are the ones being actually issued by the file system to the disk, and not serviced from cache. The "predicted operations" are the ones identified through CodeMRI. The graph on the right shows the individual contribution of different micro-workloads towards the total runtime, as predicted by CodeMRI.*

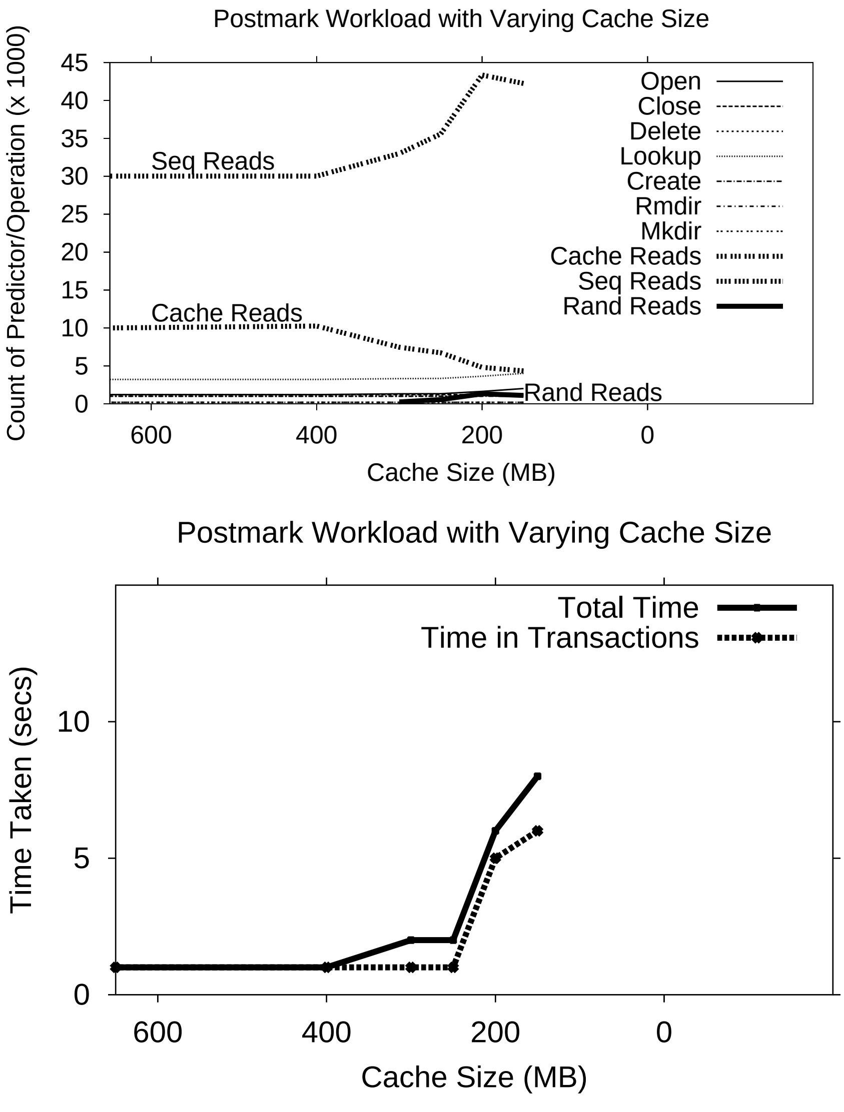


Figure 5.8 **PostMark with Decreasing Cache Size.** *The postmark configuration is 1000 files, 200 sub-directories, 4K block size, 1000 transactions, with other parameters as default. CodeMRI accurately deconstructs the effective workload and the total runtime is in accordance with the predicted workload. Important micro-workloads are in thicker lines.*

CodeMRI thus not only deconstructs workloads accurately, but the deconstructed workload is useful in predicting performance. We find that the actual runtime of the workload is in accordance with the predicted workload.

We next deconstruct a popular file-system benchmark, PostMark [70]. Figure 5.8 shows the breakdown of PostMark's workload under varying cache sizes. In the top graph, as the size of cache decreases, CodeMRI is able to identify the effect on the workload; the predicted operations contain fewer cached reads and the fraction of random reads starts to increase as well. In the bottom graph, we see the correspondence of the increased random reads and fewer cached reads on the runtime. The total runtime of the benchmark is proportional to the predicted workload, making it a useful performance indicator.

5.2 Future Work

Creating useful synthetic benchmarks is a hard problem. We have presented our first steps in building CodeMRI – a tool that enables the construction of realistic synthetic benchmarks from real workloads and file-system traces. Our initial results in applying CodeMRI to simple workloads have been promising; we intend to continue improving its accuracy for more real-world workloads. In this section we outline possible future work in deconstructing real workloads and synthesizing their realistic equivalents using CodeMRI.

5.2.1 Workload Deconstruction

Our initial attempts in deconstructing simple workloads has demonstrated the applicability of CodeMRI. However, several challenges remain to be addressed in deconstructing more complex real workloads. We discuss some future avenues to improve CodeMRI.

First, our current implementation is meant to illustrate the benefits of CodeMRI and is not optimized for production environments. In practice, we find that the small amount of tracing doesn't slow down the system appreciably, but optimizations for performance and accuracy are certainly possible. One possible approach is to reduce the code paths that need to be instrumented. The instrumentation overhead itself can be substantially reduced by tracing only the necessary minimum predictors. Since a few unique predictors serve as accurate indicators of a workload characteristic, being selective in tracing their execution profile can improve runtime performance. In future, heuristics for predictor selection can be developed and applied.

Once the size of the instrumented code base is reduced, opportunities exist for improving the accuracy of CodeMRI's deconstruction by collecting more information per instrumentation point. For example, CodeMRI currently does not trace the different flows possible within a function body. In practice, we find that at least in open-source systems, the code base is fairly modular and most of the functionality is broken down into separate routines. However, depending on predicate values, the same function might behave in different ways and in the extreme case, have all the code for a single functionality contained within. Fine-grained instrumentation for each predictor would allow us to capture these flows within a function.

Second, CodeMRI currently relies on information obtained through runtime profiling. An orthogonal approach would be to make greater use of static analysis techniques. Static analysis alone would not capture all possible regimes of operation, but together with runtime tracing, can provide stronger guarantees on code coverage during the execution of a particular workload.

Currently, CodeMRI is oblivious to the task performed by a set of predictors. It operates solely on matching a given workload with sets of micro-workloads without understanding the overall nature of the workload. Static analysis has the potential to capture semantic information about a particular predictor function which can help CodeMRI understand at a higher level what the workload is trying to get done. This information will be invaluable later on in constructing a synthetic workload which matches the desired real workload not only in terms of the execution profile, but also in its "intent".

Third, in its current form, CodeMRI needs source code for analysis, which can somewhat limit its scope. However, there is nothing fundamentally limiting CodeMRI to require instrumented source code, and in future work, CodeMRI can be made to work on executables. In order to collect the execution profile of a workload, tools such as Kerninst [143] can be used. These tools have the advantage of tracing unmodified binaries, without requiring source code access.

Fourth, the degree of domain knowledge needed to apply CodeMRI especially to a broader domain than file systems, is of particular concern. While understanding the domain is critical to constructing a benchmark that is representative of real workloads, obtaining the necessary domain knowledge can be challenging, particularly for new systems.

One approach is to use standard tools, such as strace, to understand a typical system better before building the benchmark. However, it is important to profile a number of existing systems so that the benchmark is generic, and applicable to new systems. In future, we need to extend CodeMRI to work well without explicit dependence on domain knowledge for new systems.

Finally, factors such as configuration parameters, hardware settings and real-time traffic can also affect the performance of a system. Large and important server applications, such as DB2, are quite complicated, and their performance or workload patterns are dependent not only on the application, but also on these external factors. These factors cannot be captured by analyzing the application source code alone. In future, CodeMRI needs to be able to identify the contribution to performance from non-workload elements in the running system.

In addition, in order to minimize runtime variability due to concurrent activity and non-reproducible events (e.g., interrupts), CodeMRI needs to be resilient to noise and be able to filter out the contribution of all factors extraneous to the workload. In future, we can explore use of statistical techniques similar to ones used in bug isolation [78] to improve accuracy and stability of predictions.

5.2.2 Workload Synthesis

The eventual goal of workload deconstruction is to provide enough information to construct realistic synthetic benchmarks. The microprofiles previously deconstructed now need to be used in automatically synthesizing an equivalent workload. Such a synthesis is not straightforward for several reasons. In addition to building the microprofiles, we now need to preserve timing and ordering information. Dependencies in I/O requests also need to be preserved. Although accurately determining the dependence between the timing of I/Os can be a problem for trace-driven emulations, it can be a challenge for CodeMRI as well, because micro-workloads also need to be grouped together to stress the system, at a different granularity.

Combining the microbenchmarks, in general, to make up the final benchmark is perhaps the greatest challenge in making CodeMRI useful. The question we seek to ask in future is whether any standard mathematical techniques can be applied to compose benchmarks. One approach would be to use some type of standard structure or algebra to describe the possible ways of combining the micro-workloads, since this may allow the reconstruction problem to be converted into an optimization problem. However, one requirement of this approach is that the different parts of the model, in this case, the micro-workloads, be composable, which is not always the case; for example, if one micro-workload is a foreground write, while another is a daemon that periodically writes out dirty buffers (*e.g.*, pdflush).

The reconstruction of workloads from the micro-workloads is complex because of the interactions of the different components of a file-system, especially in multi-threaded environments. One limitation of the method of deconstruction of workloads used to create the micro-workloads in the current approach is its simplified assumption of the existence of linear models to select the subset of predictors. Multiple threads, caching and applications may create non-linear dependencies in the workload functions that are exercised. In future, these restrictions need to be relaxed to make CodeMRI more broadly applicable in constructing composable benchmark workloads.

5.3 Related Work

In CodeMRI we leverage system profiling and file system domain knowledge to understand internal system behavior during execution of real workloads, and use that to create synthetic equivalents of the workload. Several tools already exist for instrumenting and profiling systems, such as, Kerninst [143], Dtrace [26], IBM's Rational PurifyPlus [108] and gprof [56]. These include features for memory corruption detection, application performance profiling, code coverage analysis and threading bugs. For our analysis, we needed a simple tracing functionality and the level of instrumentation provided by static instrumentation was sufficient since almost all of CodeMRI's logic lies outside of the tracing infrastructure. In the future, more sophisticated profiling tools can be integrated.

An alternate approach to CodeMRI as previously mentioned would be to use tools such as strace [140] to collect system calls for real applications and replay the trace. This alone will not be useful, since similar calls through the API can end up exercising the file system in very different ways and have radically different performance due to effects of caching and prefetching.

A more effective solution will be to obtain both system call and disk traces to account for file system policies and mechanisms. But there is a limitation to that approach as well. First, it is no less intrusive as compared to CodeMRI. Second, correlating strace and disk trace information is not entirely straightforward due to timing issues, especially in presence of buffering and journaling. Furthermore, the disk I/O might be reordered or delayed, and be affected by file system daemons such as `pdflush`. Semantic Block Analysis [105] is another means to infer file system level behavior, but requires detailed file system knowledge. CodeMRI has the added advantage of being oblivious of the file system in question.

Similar to our work, performance debugging of complex distributed systems [11, 19, 30] also uses tracing at various points to infer causal paths, diagnose and tune performance bottlenecks, and even to detect failures using runtime path analysis. In addition, a number of tools have been developed to understand, deconstruct and debug complex software systems such as Simpoint [127] and Shear [39]. Delta debugging is another technique that uses an automated testing framework to compare program runs and access the state of an executable program to prove the causes of program failures [160].

Mesnier *et al.* have proposed "relative fitness" models for predicting performance differences between a pair of storage devices [86]. A relative model captures the workload-device feedback, and the performance and utilization of one device can be used in predicting the performance of another device. This shifts the problem from identifying workload characteristics to device characteristics. In the future, it will be interesting to explore the use of CodeMRI together with relative fitness models.

Finally, in outlining the limitations of current benchmark workloads, we benefited from opinions expressed in previously published position papers on systems benchmarking [92, 124].

Chapter 6

Conclusions and Future Work

In this chapter, we first summarize each of the three components of our work, then discuss general lessons learned, and finally outline directions for future research.

6.1 Summary

In this section we summarize the important contributions of this dissertation. First, we discuss our findings from the five-year study of file-system metadata, and demonstrate its necessity in generating representative and reproducible benchmarking state. Second, we review our methodology to allow large, real benchmark workloads to be run in practice with a modest storage infrastructure. Finally, we discuss challenges in generating representative, reproducible, and practical file-system benchmark workloads, and present our initial steps in creating an automated workload synthesizer.

6.1.1 Representative and Reproducible Benchmarking State

Developers of file systems and data-centric applications frequently need to make assumptions about the properties of file-system images. For example, file systems and applications can often be optimized if they know properties such as the relative proportion of metadata to data or the frequency of occurrence of various file types, in representative file systems. Getting pertinent information about representative file systems requires access to usage information about file-system metadata.

To develop an understanding of file-system metadata in desktop computers, we analyzed the static and longitudinal properties of metadata by conducting a large-scale study of file-system contents. To generate file-system images that are representative of such real-world characteristics in a reproducible fashion, we developed a statistical framework that allows one to incorporate realistic characteristics of file system metadata and file data.

6.1.1.1 Characteristics of File-System Metadata

For five years, from 2000 to 2004, we collected annual snapshots of file-system metadata from over 60,000 Windows PC file systems at Microsoft Corporation. We used these snapshots to study temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity and consumption, and degree of file modification. We presented a generative model that explains the namespace structure and the distribution of directory sizes. We found significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities. We gave examples of consequent lessons for designers of file systems and related software. Here, we summarize again the important observations from our study:

- The space used in file systems has increased over the course of our study, not only because mean file size has increased (from 108 KB to 189 KB), but also because the mean number of files has increased (from 30K to 90K).

- Eight file-name extensions account for over 35% of files, and nine file-name extensions account for over 35% of the bytes in files. The same sets of extensions including `cpp`, `dll`, `exe`, `gif`, `h`, `htm`, `jpg`, `lib`, `mp3`, `pch`, `pdb`, `pst`, `txt`, `vhd`, and `wma` have remained popular for many years.

- The fraction of file-system content created or modified locally has decreased over time. In the first year of our study, the median file system had 30% of its files created or modified locally, and four years later this percentage was 22%.
- Directory size distribution has not notably changed over the years of our study. In each year, directories have had very few sub directories and a modest number of entries. 90% of them have had two or fewer sub directories, and 90% of them have had 20 or fewer total entries.
- The fraction of file system storage residing in the namespace subtree meant for user documents and settings has increased in every year of our study, starting at 7% and rising to 15%. The fraction residing in the subtree meant for system files has also risen over the course of our study, from 2% to 11%.
- File system capacity has increased dramatically during our study, with median capacity rising from 5 GB to 40 GB. One might expect this to cause drastic reductions in file system fullness, but instead the reduction in file system fullness has been modest. Median fullness has only decreased from 47% to 42%.
- Over the course of a single year, 80% of file systems become fuller and 18% become less full.

Our measurements revealed several interesting properties of file systems and offered useful lessons. While the present study offers a thorough analysis of metadata representative of one corporate desktop environment, it is anything but representative of the many different usage scenarios in the field. We certainly hope that it encourages others to collect and analyze data sets in different environments, contributing them to a public repository such as the one maintained by SNIA [133]; our understanding of file-system metadata properties and trends from this study certainly provided the impetus, and much of the basis for our work on Impressions.

6.1.1.2 Generating File-System Benchmarking State

Motivated by the knowledge gained from our metadata study, we developed *Impressions*, a framework to generate statistically accurate file-system images with realistic metadata and content. Impressions is flexible, supporting user-specified constraints on various file-system parameters using a number of statistical techniques to generate consistent images. In this dissertation, we presented the design, implementation and evaluation of Impressions.

Developers frequently require representative file-system images to test a new feature or make comparisons with existing ones. Previously no tool existed that allowed creation of file-system images with accurate approximation of real data and attention to statistical details; developers often ended up writing limited in-house versions for generating test cases, something that Impressions strives to standardize.

Impressions makes it extremely easy to create both controlled and representative file-system images. First, Impressions enables developers to tune their systems to the file system characteristics likely to be found in their target user populations. Second, it enables developers to easily create images where one parameter is varied and all others are carefully controlled; this allows one to assess the impact of a single parameter. Finally, Impressions enables different developers to ensure they are all comparing the same image; by reporting Impressions parameters, one can ensure that benchmarking results are reproducible.

Impressions also proved useful in discovering application behavior. For example, we found that Google Desktop for Linux omits certain content from indexing based on specified hard values for file depth in the file system tree; content omission also happens for varying organizations of the file system tree. This strange behavior further motivates the need for a tool like Impressions to be a part of any application designer's toolkit. We believe that instead of arbitrarily specifying hard values, application designers should experiment with Impressions to find acceptable choices for representative images. We note that Impressions is useful for discovering these application assumptions and for isolating performance anomalies that depend on the file-system image.

In informal conversations with developers of open-source software, and with researchers and academicians, we found that Impressions was of immediate benefit to the file and storage community. In particular, open-source development projects on desktop search [21] and data backup [156], commercially available concurrent visioning file system [71] and software for management of SAN storage in vitalized environments [81], and several academic research groups [49, 72, 76, 109, 123, 151, 163] have expressed interest in the Impressions framework for conducting their testing and benchmarking; we hope and expect it to evolve into a useful platform for benchmarking.

6.1.2 Practical Benchmarking for Large, Real Workloads

Motivated by our own experience (and consequent frustration) in doing large-scale, realistic benchmarking, we developed Compressions, a “scale-down” benchmarking system that allows one to run large, complex workloads using relatively smaller storage capacities. Compressions makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system by transparently scaling down the storage capacity required to run the workload.

In practice, realistic benchmarks (and realistic configurations of such benchmarks) tend to be much larger and more complex to set up than their trivial counterparts. File system traces (*e.g.*, from HP Labs [113]) are good examples of such workloads, often being large and unwieldy. In many cases the evaluator has access to only a modest infrastructure, making it harder still to employ large, real workloads.

We started with the hypothesis that evaluators would appreciate having the means to run benchmarks without having to spend time, effort and expense on continually upgrading their storage capacities for benchmarking. In building Compressions, we soon realized that since our compressed version could run much faster than the native workload execution on the original storage system, we could actually speedup the runtime of the benchmark itself without sacrificing the authenticity of the benchmark experiment. The storage model within Compressions interposes on all I/O requests and computes the time taken to run the benchmark on the original system. We believe this is a useful feature since most large benchmarks also take a long time to run (often several hours to days) which reduces their practical application.

There were two primary challenges we faced in developing Compressions. First, having a model instead of the real system required us to precisely capture the behavior of the entire storage stack with all its complex dependencies. Since our model is in line with workload execution, it needed to be extremely fast so as to not slow down the workload itself. Second, Compressions discards writes to file data and thus needs to synthetically generate it for subsequent reads while adhering to the semantics of the application issuing the I/Os.

We have addressed both these challenges in Compressions. First, our model framework is sufficiently lean while still being fairly accurate. In building the model, we stayed away from any table-based approaches that required us to maintain runtime statistics. Instead wherever applicable, we adopted and developed analytical approximations that did not slow the system down and still maintained accuracy. Second, synthetic content generation in Compressions leverages our prior work in building Impressions; we use a combination of built-in content generation modules with some hints from the application where necessary to generate suitably tailored file content.

With storage capacity growth showing no signs of leveling off, we expect the trend towards larger benchmarks and application working sets to continue. Compressions can serve as an effective substitute to capacity upgrades and provide a test bed for answering questions about performance that were previously infeasible.

6.1.3 Representative, Reproducible and Practical Benchmark Workloads

Apart from the file-system image, benchmark workloads are the other important requirement for a benchmark. To evaluate the performance of a file and storage system, developers have a few different options including real applications, microbenchmarks of application kernels, trace replay, and synthetic workloads; each choice comes with its own set of advantages and disadvantages.

In a survey of benchmark workloads conducted by us, we found that synthetic benchmarks are the ones most popular in the file systems community. Synthetic workloads are designed to stress file systems appropriately, containing a mix of POSIX file operations that can be relatively scaled to stress different aspects of the system; some examples include IOZone [98], SPECsfs97 [157], SynRGen [44], fstress [12], and Chen's self-scaling benchmark [31]. The major advantages of synthetic applications is how simple they are to run and that they can be adapted as desired. However, the major drawback of synthetic workloads is that they may not be representative of any real workloads that users care about.

Given the popularity of synthetic benchmark workloads, we believed that an ideal benchmark for file and storage systems combines the *ease of use* of a synthetic workload with the *representativeness* of a real workload; our hypothesis was that if two workloads execute roughly the same set of function calls within the file system, that they will be roughly equivalent to one another.

While creating representative benchmark workloads is not an entirely solved problem, significant steps have been taken by others towards this goal. Empirical studies of file-system access patterns [17, 58, 100] and file-system activity traces [113, 133] have had led to work on synthetic workload generators [12, 44] and methods for trace replay [14, 85]. However, automated workload synthesizers are hard to write. Current methods for creating synthetic benchmark workloads were largely based on the benchmark writer's interpretation of the real workload, and how it exercised the system API.

We believed that in order to create an equivalent synthetic workload for file and storage systems, one must mimic not the system calls, but the *function calls* exercised during workload execution, in order to be *functionally equivalent*. With this approach in mind, we developed CodeMRI, a tool to create synthetic benchmarks that are functionally equivalent to and representative of real workloads.

Our initial experiences with CodeMRI were positive; we were able to deconstruct somewhat complex workloads into simpler micro-workloads and also accurately predict performance. CodeMRI's eventual success depends on its ability to deconstruct complex, real workloads, and more importantly to be able to create an equivalent synthetic workload; we outlined steps for future work on CodeMRI in Chapter 5.

6.2 Lessons Learned

In this section, we briefly discuss the general lessons we learned while working on this thesis; we classify them into two broad categories:

Principles and Design

Benchmarks are crucial, yet overlooked

Benchmarks are crucial to the file and storage system community but the amount of time and effort spent in developing benchmarking technologies is not commensurate. It is often concluded that research on benchmarking is focused on measurement and not on system building, and thus not enough system developers consider participating.

While working on this thesis, we found that several challenges in benchmarking require new systems to be designed and built for carrying out the benchmarking, much in the same way a new file or storage system is built.

Real-world statistics are hard to get, but invaluable

Statistics on real-world usage of file and storage systems is hard to obtain. Often the underlying data is hard to collect as system administrators are paranoid about setting up probes in production systems (perhaps rightfully so); at other times, the data is relatively easy to collect, but privacy concerns prevent it from being publicly visible.

By provisioning the storage systems with externally visible probe points, and by developing techniques to ensure that the collected data is easy to share without compromising privacy, we believe storage developers can gain access to valuable statistics on how their systems actually get used. We learnt a lot from our five-year study of file-system metadata and encourage others to follow suit.

We can all be better practitioners

Current benchmarks and tools for benchmarking are not perfect, but as users of these technologies, we are responsible for judiciously using them to get the best possible results.

By using relevant configurations for benchmarks, by reporting all benchmark parameters in detail, and by making the resulting software and experimental data publicly available, we can improve the quality of benchmarking even with the available resources.

Ease of use cannot be underestimated

However naive it might seem, ease of use is often a critical factor in the popularity and adoption of a benchmark; we have thus built our tools with ease of use as a primary objective.

Long-term Sustainability

No benchmark can be future proof

Technology trends, new applications, and changes in usage patterns can all render a benchmark irrelevant over time. Improvements need to be periodically applied to any benchmark to account for such changes; benchmarks thus must be made freely available with the source code.

Postmark is a classic example of a benchmark that when written was perhaps an adequate representation of mail-server workloads, but is inadequate by any current standards. In order to keep Postmark relevant, one needs to account for changes in mail-server usage, the different storage schema available for storing emails, and sizes for the email files.

Community involvement is critical

For a benchmark to be authentic and remain such, community involvement is critical. The file and storage community needs to define a broader set of guidelines on how to use a given benchmark including appropriate set of configuration parameters, develop best practices to be followed while benchmarking, and most importantly encourage participation of the community in following the standardized guidelines; more details on some recent initiatives are available elsewhere [28, 75, 146].

6.3 Future Work

In this section, we outline various avenues for future research. First, we discuss possible studies of various file-system attributes. Second, we outline future extensions to Impressions in generating file system and disk state. Third, we discuss extending Compressions to include storage models for storage devices such as SSDs. Finally, we propose our vision towards a unified benchmarking platform.

6.3.1 Empirical File-System Studies

Future efforts to understand different properties of file-system metadata, file content, and on-disk layout can augment our own findings and answer questions that our study does not focus on.

File content is an interesting and complex attribute to study. In our work we have used analytical language models and dictionary-based approaches for generating natural language content. A large-scale study of file content would certainly be useful to extend Impressions and Compressions, particularly to address questions on data redundancy, content-addressability, and compressability.

Another interesting study would be to look at fragmentation in real systems. While fragmentation can definitely affect performance, how much fragmentation really is there in current disk-based systems? How would this change with the advent of solid-state storage that does not suffer from fragmentation in the same way as a rotating disk?

The findings from our metadata study are bound to get outdated in the future, if not already. Repeats of such a study will help keep the information up-to-date. While our study focused on one corporate environment, in the future, similar studies can be performed in other environments of interest such as data-centers, university laboratories and home users.

6.3.2 Generating Realistic File System and Disk State

Impressions currently provides minimal support for laying out the files on disk in a realistic fashion; in particular, fragmented file systems are generated by creating and deleting temporary files during the creation of the long-term files. In future, more work can be done to induce realistic fragmentation by considering out-of-order file writes, or writes with long delays between them. Another factor affecting fragmentation is that, in many cases, the majority of the file system is created all at once at file system creation time (e.g., during OS installation), while the rest is written more slowly and burstily. Dividing file system creation into two phases with separate parameters would be helpful.

One more factor for fragmentation is that many file systems allocate blocks differently if the writes come all at once, as in quick creation of file system images, or trickle slowly, as a result of delayed allocation. Approximation of this effect can be achieved by interspersing `syncs` during creation. All this can perhaps be part of a companion tool to Impressions, aptly named *Depressions*.

Impressions itself can be extended in many ways, we list three important ones here. First, Impressions currently supports creation of local file systems on a single machine; in future, support for creation of distributed file systems and file systems for other storage devices such as SSDs can also be included. Second, content generation in Impressions is rather simplistic and does not capture application requirements for more complex file data and file types; support for more realistic content generation can be developed in the future, somewhat along the lines of the profile store and RAW cache in Compressions. Third, a desirable feature in Impressions is to allow it to traverse an existing file system and generate the necessary statistics on-the-fly to be used later on, allowing developers to easily share the details of their file-system images.

6.3.3 Benchmarking Storage Beyond Hard Drives

Although our work on Impressions and Compressions has focused on developing systems that are applicable for rotating hard drives and file systems developed for these devices, it is not fundamentally limited to benchmark hard drives alone.

In future, Compressions can certainly be augmented with models for more complex storage systems such as RAID arrays and storage clusters. The underlying device itself can be replaced by a flash-based solid state drive (SSD); a model for an SSD can be built into Compressions by leveraging our previous work on understanding the properties of such devices [10]. The policies for the metadata layout and the synthetic generation of file content in Compressions can also be extended to include environments that use SSDs instead.

6.3.4 Holistic Storage Benchmarking Platform

In this dissertation we designed and developed three systems, Impressions, Compressions, and CodeMRI, with the goal of simplifying file and storage benchmarking; we believe our attempts have been fruitful and encouraging. Our vision is to combine these, or other incarnations of such systems into a unified benchmarking platform that serves as a community resource. While this vision might be too grand to achieve in the short term, we believe it is in the right direction.

LIST OF REFERENCES

[1] Ashraf Aboulnaga, Jeffrey F. Naughton, and Chun Zhang. Generating synthetic complex-structured xml data. In *WebDB*, pages 79–84, 2001.

[2] Atul Adya, William Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John Douceur, Jon Howell, Jacob Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, Boston, MA, December 2002.

[3] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards Realistic File-System Benchmarks with CodeMRI. In *First Workshop on Hot Topics in Measurement and Modeling of Computer Systems (ACM HotMetrics '08)*, Annapolis, MD, June 2008.

[4] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards realistic file-system benchmarks with codemri. *SIGMETRICS Perform. Eval. Rev.*, 36(2):52–57, 2008.

[5] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Compressions: Enabling File System Benchmarking at Scale. In *Preparation for submission to a top-tier systems conference*, 2009.

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

[7] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata: Microsoft longitudinal dataset. `http://iotta.snia.org/traces/list/Static`.

[8] 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 Conference on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[9] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. *ACM Transactions on Storage*, 3(3), October 2007.

[10] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the Usenix Annual Technical Conference (USENIX '08)*, Boston, MA, June 2008.

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

[12] Darrell Anderson and Jeff Chase. Fstress: A flexible network file service benchmark. In *TR, Duke University, May 2002.*

[13] Eric Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.

[14] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[15] Apple. Technical Note TN1150. http://developer.apple.com/technotes/tn/tn1150.html, March 2004.

[16] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 43–56, Banff, Canada, October 2001.

[17] 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.

[18] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 151–160, Madison, WI, June 1998.

[19] Paul Barham, Rebecca Isaacs, Richar Mortier, and Dushyanth Narayanan. Magpie: Real-Time Modeling and Performance-Aware Systems. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[20] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[21] Beagle Project. Beagle Desktop Search. `http://www.beagle-project.org/`.

[22] J. Michael Bennett, Michael A. Bauer, and David Kinchlea. Characteristics of files in NFS environments. In *Proceedings of the 1991 ACM SIGSMALL/PC Symposium on Small Systems*, pages 33–40, Toronto, Ontario, June 1991.

[23] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

[24] Jeff Bonwick. Zfs: The last word in file systems. Available at `http://www.opensolaris.org/os/community/zfs/docs/zfs\_last.pdf`.

[25] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from risc machines: Generation and analysis. Technical Report 89/14, Digital Equipment Western Research Laboratory Research Report, 1989.

[26] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, pages 15–28.

[27] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 12–21, New York, NY, USA, 1993. ACM.

[28] Erez Zadok (Chair). File and storage systems benchmarking workshop. UC Santa Cruz, CA, May 2008.

[29] Greg Chapman. Why does Explorer think I only want to see my documents? Available at `http://pubs.logicalexpressions.com/Pub0009/LPMArticle.asp?ID=189`.

[30] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[31] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*.

[32] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[33] James Cipar, Mark D. Corner, and Emery D. Berger. Tfs: a transparent file system for contributory storage. In *FAST '07*, pages 28–28, Berkeley, CA, USA, 2007. USENIX Association.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001. 35.5: The subset-sum problem.

[35] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, Boston, MA, December 2002.

[36] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36, 2002.

[37] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 120–132, New York, NY, USA, 2003. ACM.

[38] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[39] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 59–71, Boston, Massachusetts, October 2004.

[40] David J. DeWitt. The wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[41] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 59–70, Atlanta, GA, May 1999.

[42] Allen B. Downey. The structural cause of file size distributions. In *Proceedings of the 2001 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 328–329, Cambridge, MA, June 2001.

[43] Allen B. Downey. The structural cause of file size distributions. In *Ninth MASCOTS'01*, Los Alamitos, CA, USA, 2001.

[44] Maria R. Ebling and M. Satyanarayanan. Synrgen: an extensible file reference generator. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, New York, NY, 1994.

[45] Michael Eisler, Peter Corbett, Michael Kazar, Daniel S. Nydick, and Christopher Wagner. Data ontap gx: a scalable storage cluster. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 23–23, Berkeley, CA, USA, 2007. USENIX Association.

[46] Kylie M. Evans and Geoffrey H. Kuenning. A study of irregularities in file-size distributions. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, San Diego, CA, July 2002.

[47] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton and Company Inc, New York, NY, first edition, 1978.

[48] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24, 2002.

[49] Simson Garfinkel. 'Request for using Impressions.'. Personal Communication, 2009.

[50] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*.

[51] 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.

[52] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.

[53] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic File Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[54] Google Corp. Google Desktop for Linux. `http://desktop.google.com/linux/index.html`.

[55] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99: Third symposium on Operating Systems Design and Implementation*, 1999.

[56] Gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html. 1998.

[57] GraphApp. GraphApp Toolkit. `http://enchantia.com/software/graphapp/`.

[58] Steven D. Gribble, Gurmeet Singh Manku, Drew S. Roselli, Eric A. Brewer, Timothy J. Gibson, and Ethan L. Miller. Self-similarity in file systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 141–150, Madison, WI, June 1998.

[59] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate Storage Emulation. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[60] P. G. Guest. *Numerical Methods of Curve Fitting*. Cambridge University Press, Cambridge, UK, 1961.

[61] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 60–71, Madison, WI, June 2005.

[62] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: time-warped network emulation. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[63] John L. Hennessy and David A. Patterson, editors. *Computer Architecture: A Quantitative Approach, 3rd edition*. Morgan-Kaufmann, 2002.

[64] Kenneth Houkjaer, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[65] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System.

[66] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. Physical File System Backup. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[67] Gordon Irlam. Unix file size survey – 1993. Available at `http://www.base.com/gordoni/ufs93.html`.

[68] John McCutchan and Robert Love. inotify for linux. `http://www.linuxjournal.com/article/8478`.

[69] Jonathan Corbet. LWN Article: SEEK_HOLE or FIEMAP? `http://lwn.net/Articles/260795/`.

[70] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[71] Pramod Khakural. 'Request for using Impressions.'. Personal Communication, 2009.

[72] Richa Khandelwal. 'Request for using Impressions.'. Personal Communication, 2009.

[73] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[74] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report TR94-220, Dartmouth College, 1994.

[75] Geoff Kuenning and Bruce Worthington. Trace collection and sharing. In *USENIX FAST BIRDS-OF-A-FEATHER SESSION*, San Jose, CA, February 2008.

[76] Vivek Lakshmanan. 'Request for using Impressions.'. Personal Communication, 2009.

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

[78] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI '05*.

[79] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. http://www.namesys.com/txn-doc.html, 2002.

[80] Hosam M. Mahmoud. Distances in random plane-oriented recursive trees. *Journal of Computational and Applied Mathematics*, 41:237–245, 1992.

[81] Angelo Masci. 'Request for using Impressions.'. Personal Communication, 2009.

[82] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world ai systems. *Artificial Intelligence for Applications, Conference on*, 0:87, 1995.

[83] 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.

[84] Michael Mesnier, Eno Thereska, Gregory R. Ganger, Daniel Ellard, and Margo Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, New York, NY, May 2004.

[85] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. trace: parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[86] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[87] Microsoft. SetFileTime. Available at MSDN, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50lrfsetfiletime.asp`.

[88] Microsoft. Sql server 2000 resource kit. In *Microsoft TechNet*, October 2006.

[89] Stan Mitchell. *Inside the Windows 95 file system*. O'Reilly and Associates, 1997.

[90] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. In *Internet Mathematics*, 2002.

[91] Michael Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics*, 1(3):305–333, 2004.

[92] Jeffrey C. Mogul. Brittle metrics in operating systems research. In *HotOS '99*.

[93] Mplayer. The MPlayer movie player. `http://www.mplayerhq.hu/`.

[94] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Software—Practice and Experience*, 14(4):365–368, April 1984.

[95] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 174–187, Banff, Canada, October 2001.

[96] Myers Carpenter. Id3v2: A command line editor for id3v2 tags. `http://id3v2.sourceforge.net/`.

[97] NIST. Text retrieval conference (trec) datasets. http://trec.nist.gov/data, 2007.

[98] William Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[99] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[100] 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.

[101] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–24, Orcas Island, WA, December 1985.

[102] Yoann Padioleau and Olivier Ridoux. A logic file system. In *USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

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

[104] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, Texas, June 2003.

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

[106] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[107] Bartosz Przydatek. A Fast Approximation Algorithm for the Subset-sum Problem. *International Transactions in Operational Research*, 9(4):437–459, 2002.

[108] PurifyPlus. http://www-306.ibm.com/software/awdtools/purifyplus/. 2005.

[109] Alex Rasmussen. 'Request for using Impressions.'. Personal Communication, 2009.

[110] Hans Reiser. Three reasons why ReiserFS is great for you. Available at `http://www.namesys.com/`.

[111] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[112] Richard McDougall. Filebench: Application level file system benchmark.

[113] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

[114] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[115] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

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

[117] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[118] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *USENIX Winter*, pages 405–420, 1993.

[119] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[120] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 96–108, Pacific Grove, CA, December 1981.

[121] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[122] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[123] Margo Seltzer. 'Request for using Impressions.'. Personal Communication, 2009.

[124] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *HotOS*, 1999.

[125] Sam Shah and Brian D. Noble. A study of e-mail patterns. *Softw., Pract. Exper.*, 37(To Appear), 2007.

[126] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using provenance to aid in personal file search. In *USENIX Annual Technical Conference*, pages 171–184, Santa Clara, CA, June 2007.

[127] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS '02*.

[128] Tracy F. Sienknecht, Rich J. Friedrich, Joe J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, 20(1–3):3–25, May 1994.

[129] Bengt Sigurd, Mats Eeg-Olofsson, and Joost van de Weijer. Word length, sentence length and frequency – Zipf revisited. *Studia Linguistica*, 58(1):37–52, 2004.

[130] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[131] Keith Smith and Margo Seltzer. File layout and file system performance. Technical Report TR-35-94, Harvard University, 1994.

[132] Keith Smith and Margo I. Seltzer. File System Aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[133] SNIA. Storage network industry association: Iotta repository. http://iotta.snia.org, 2007.

[134] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilie Shao, Chi Zhang, Wlisha Ziskind, and Arvind Krishnamurthy. Segank: A Distributed Mobile Storage System. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 239–252, San Francisco, California, April 2004.

[135] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[136] David A. Solomon. *Inside Windows NT*. Microsoft Press, 2nd edition, 1998.

[137] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *SOSP*, pages 119–132, Brighton, United Kingdom, October 2005.

[138] SPC. Storage performance council. http://www.storageperformance.org/, 2007.

[139] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[140] strace. http://linux.die.net/man/1/strace. 2008.

[141] 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.

[142] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[143] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 117–130, New Orleans, Louisiana, February 1999.

[144] Doug Thain, John Bent, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 12)*, pages 152–161, Seattle, Washington, June 2003.

[145] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. Accepted for publication, ETA February 2008.

[146] Avishay Traeger and Erez Zadok. How to cheat at benchmarking. In *USENIX FAST BIRDS-OF-A-FEATHER SESSION*, San Francisco, CA, February 2009.

[147] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[148] Transaction Processing Council. TPC Benchmark C Standard Specification, Revision 5.2. Technical Report, 1992.

[149] Transaction Processing Council. TPC Benchmark H Standard Specification, Revision 2.8. Technical Report, 1992.

[150] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[151] Ruey-Yuan Ryan Tzeng. 'Request for using Impressions.'. Personal Communication, 2009.

[152] Kaushik Veeraraghavan, Andrew Myrick, and Jason Flinn. Cobalt: separating content distribution from authorization in distributed file systems. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 29–29, Berkeley, CA, USA, 2007. USENIX Association.

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

[154] Glenn Weinberg. The Solaris Dynamic File System. http://members.visi.net/~thedave/sun/DynFS.pdf, 2004.

[155] Lars Wirzenius. 'Genbackupdata: tool to generate backup test data'. `http://braawi.org/genbackupdata.html`, 2009.

[156] Lars Wirzenius. 'Request for using Impressions.'. Personal Communication, 2009.

[157] M. Wittle and Bruce E. Keith. LADDIS: The next generation in NFS file server benchmarking. In *USENIX Summer*, pages 111–128, 1993.

[158] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. Online extraction of SCSI disk drive parameters. Technical Report CSE-TR-323-96, Carnegie Mellon University, 19 1996.

[159] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Anaheim, CA, April 2005.

[160] Andreas Zeller. Isolating cause-effect chains from computer programs. In *10th ACM SIGSOFT symposium on Foundations of software engineering*, 2002.

[161] Zhihui Zhang and Kanad Ghose. yfs: A journaling file system design for handling large data sets with reduced seeking. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 59–72, Berkeley, CA, USA, 2003. USENIX Association.

[162] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. Tbbt: scalable and accurate trace replay for file server evaluation. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[163] Liu Zhuo. 'Request for using Impressions.'. Personal Communication, 2009.

APPENDIX
Analytical Distributions

A.1 Continuous Distributions

Binary log-normal, x $>$ 0:

$$f_1(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}ln2}e^{\frac{-(lgx-\mu)^2}{2\sigma^2}} \tag{A.1}$$

Inverse-polynomial of degree N, offset α, x $>$ 0:

$$f_p(x;N,\alpha) = (N-1)\alpha^{N-1}(x+\alpha)^{-N} \tag{A.2}$$

R-stage hyperexponential, x $>$ 0:

$$f_h(x;\mathcal{R},\alpha,\mu) = \sum_{i=1}^{\mathcal{R}} \alpha_i\mu_i e^{-\mu_i x} \tag{A.3}$$

Pareto, k $\leq$ x:

$$f_p(n;k,\alpha) = \alpha k^{\alpha} x^{-\alpha\ -1} \tag{A.4}$$

A.2 Discrete Distributions

Poisson, n $\geq$ 0:

$$f_p(n;\lambda) = \frac{\lambda^n e^{-\lambda}}{n!} \tag{A.5}$$

Generalized Zipf, 1 $\leq$ n $\leq$ N:

$$f_z(n;N,b,\theta) = \frac{(-1)^{\theta}\Gamma(\theta)}{\Psi^{(\theta-1)}(b+1)-\Psi^{(\theta-1)}(N+b+1)}(n+b)^{-\theta} \tag{A.6}$$

Generalized Lotka, n $\geq$ 1:

$$f_z(n;\theta) = \frac{1}{\zeta(\theta)} n^{-\theta} \tag{A.7}$$

APPENDIX
Statistical Techniques and Terminology

Some of the following are derived from standard statistics text books [47, 60].

Generative Model: Model for randomly generating observable data typically given some hidden parameters. Generative models specify a joint probability distribution over observed values.

Monte Carlo Method: Class of computational algorithms that rely on repeated random sampling to compute their results, often used when simulation is being performed by a computer due to their need for repeateded generation of pseudo-random numbers. Particularly useful when deterministic solutions are infeasible.

Null hypothesis: The hypothesis that an observed difference (such as one between observed and modeled values, or between two sets of observed values) just reflects chance variation.

Alternative hypothesis: The hypothesis that the observed difference is real, *i.e.*, the opposite of the null hypothesis.

Test statistic: Used to measure the difference between observed data and the expected data under the null hypothesis.

P-value: For a hypothesis test, the p-value is the probability computed under the null hypothesis. In other words it is the chance of getting the test statistic as extreme as or more extreme than the observed one [47].

Goodness-of Fit: Quantitative means to describe how well a statistical model distribution fits observed data. A distance metric (or *test statistic*) usually specifies the discrepancy between observed and modeled values.

Kolmogorov-Smirnov test: Goodness-of-fit test for checking whether a given distribution is not significantly different from a hypothesized distribution, used for continuous probability distributions. The test is as follows:
If $X_1, X_2, ..., X_n$ be independent and identically-distributed random variables from a continuous cumulative density function F, let F_n be the empirical cumulative density function.
$D_n(F) = sup_{x \in \mathbb{R}} |(F_n(x) - F(x)|$ is the distance between F_n and F, $\rho_\infty(F_n, F)$.
D_n is the test statistic for the K-S test and is small if the null hypothesis is true, $H_0 : F = F_0$.
Tests of the form $D_n(F_0) > c$ where null hypothesis is rejected if the D statistic is greater than c are called Kolmogorov-Smirnov tests.

Chi-Square test: Goodness-of-fit test for checking whether a given distribution is not significantly different from a hypothesized distribution, used for discrete probability distributions. The test is as follows:

Let $(O_1, O_2, ..., O_n)$ and $(E_1, E_2, ..., E_n)$ be observed and expected frequencies. The test statistic
$\chi^2 = \sum_{i=0}^{\mathcal{N}} \frac{(O_i - E_i)^2}{E_i}$, which cannot be negative. Larger values of χ^2 indicate a greater discrepancy between the observed and expected distributions. Similar to the K-S test, a p-value is computed under the null hypothesis to test whether the hypothesis can be rejected.

Linear Interpolation: For known coordinates given by (x_0, y_0) and (x_1, y_1), a linearly interpolated value for y for x $\in (x_0, x_1)$ is given by: $y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$

Heavy-tailed distribution: The distribution of a random variable X with distribution function F is said to be heavy-tailed if $\lim_{x \to \infty} e^{\lambda x} \Pr[X > x] = \infty \quad$ for all $\lambda > 0$