Multi-dimensional Workload Analysis and Synthesis for Modern Storage Systems

A Dissertation Presented

by

Vasily Tarasov

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2013

Stony Brook University
The Graduate School

Vasily Tarasov

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Dr. Erez Zadok—Dissertation Adviser
Associate Professor, Department of Computer Science

Dr. Donald Porter—Chairperson of Defense
Assistant Professor, Department of Computer Science

Dr. Michael Ferdman
Assistant Professor, Department of Computer Science

Dr. Geoffrey Kuenning
Professor, Harvey Mudd College, Department of Computer Science

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Multi-dimensional Workload Analysis and Synthesis for Modern Storage Systems

by

Vasily Tarasov

Doctor of Philosophy
in
Computer Science

Stony Brook University

2013

Modern computer systems produce and process an overwhelming amount of data at an increasing rates. The performance of storage hardware components, however, cannot keep up with the required speed at a practical cost. To mitigate this discrepancy, storage vendors incorporate many *workload-driven optimizations* in their products. Emerging applications cause workload patterns to change rapidly and significantly. One of the prominent examples is a rapid shift towards virtualized environments. Virtual machines mix I/O streams from different applications and perturb applications' access patterns. In addition, modern users demand more convenience features, such as deduplication, snapshotting, and encryption. Stringent performance requirements, changing I/O patterns, and the growing feature list increase the complexity of storage systems. The complexity of design, in turn, makes the evaluation of the storage systems a difficult task.

Storage community needs practical evaluation tools and techniques to resolve this task timely and efficiently. This thesis first explores the complexity of evaluating storage systems. Second, the thesis proposes a Multi-Dimensional Histogram (MDH) workload analysis as a basis for designing a variety of evaluation tools. I/O traces are good sources of information about real-world workloads but are inflexible in representing more than the exact system conditions at the point the traces were captured. We demonstrate how MDH techniques can accurately convert I/O traces to workload models. Historically, most I/O optimizations focused on the metadata: e.g., I/O access patterns, arrival times, read/write sizes. Increasingly, storage systems must also consider the *data* and not just the metadata. For example, deduplication systems eliminate duplicates in the data to increase logical storage capacity. We use MDH techniques to generate realistic datasets for deduplication systems. The shift from physical to virtual clients drastically changes the I/O workloads seen by Network Attached Storage (NAS). Using MDH techniques we study workload changes caused by virtualization and synthesize a set of versatile NAS benchmarks.

It is our thesis that MDH techniques are powerful for both workload analysis and synthesis. MDH analysis bridges the gap between the complexity of storage systems and the availability of practical evaluations tools.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not be possible without many people that helped me on the thorny path of a Ph.D. student. First, I would like to thank all master students that spent uncountable number of hours on helping me with every aspect of the work: Binesh Andrews, Sujay Godbole, Deepak Jain, Mandar Joshi, Atul Karmarkar, Rachita Kothiyal, Santhosh Kumar, Ravikant Malpani, Sonam Mandal, Amar Mundrakit, Karthikeyani Palanisami, Priya Sehgal, Gyumin Sim, and Sagar Trehan. I especially would like to thank Santhosh Kumar and Amar Mundrakit who were involved in Trace2Model and Deduplication projects.

I also would like to thank truly exceptional students from Harvey Mudd College: Will Buik, Garry Lent, Jack Ma, and Megan O'Keefe. I'm very thankful to Will Buik and Jack Ma who—instead of relishing in sunny California—spent their 2011 summer in FSL working closely with me. I hope they enjoyed their experience in the lab as much as I enjoyed working with them. Without all these undergraduate students' help, this work would have been much harder.

I deeply thank Nikolay Joukov who graciously agreed to mentor an inexperienced student during his first internship at IBM T.J. Watson Research Center. Next two internships I spent at IBM Research—Almaden. Dean Hildebrand, Anna Povzner, and Renu Tewari are the people to thank there. I'm especially thankful to Dean, my mentor, who was always open for a discussion and did every single thing for a productive internship. Thanks to his efforts, a very successful collaboration between IBM Almaden's storage group and the FSL continued beyond that summer. I also should thank Dean and Renu for supporting me as a nominee for IBM Ph.D. Fellowship. This not only made me a proud holder a fellow title, by also eased the financial part of my everyday life.

Philip Shilane from EMC was helping us to understand many deduplication nuances. He gave us an access to the real-world datasets, which allowed us to complete the deduplication part of this thesis. His wise reasoning can be found in many places throughout the thesis.

I would like to thank three persons from academia. First, I thank Margo Seltzer from Harvard University who was working with us on the file system benchmarking project. Her passion about a thorough understanding of storage systems performance is infectious. Second, Geoff Kuenning from Harvey Mudd College contributed a lot to this thesis. Being a collaborator in most of my projects, he was a faultless gauge of what should be kept and what should be left out when it comes to deciding which research directions to pursue in a limited time. I adore his deep love of English language and I am sure that I made him suffer multiple times when he was revising (and often completely rewriting) my drafts. Finally, there is not enough space in this section to express all the thanks to my adviser—Erez Zadok. I was a lucky beggar when he, somewhat accidentally, agreed to be my adviser. Rarely there are advisers that are so deeply involved in the projects, guide students trough the years so professionally, demonstrate proper research techniques, and mercilessly exterminate inefficient practices.

Finally, I want to thank my friends in and outside the lab. Especially Pradeep Shetty, a Master student who has already graduated, and Zhichao Li and Ming Chen, Ph.D. students who still have the graduation path in front of them. My friends—Tatsiana Mironava, Yury Puzis, and Eugene Borodin are the people that made my leisure fun. Thanks to them, I was rarely bored these years.

No good research is possible without financial support. In addition to the aforementioned two IBM Ph.D. Fellowships, this work was supported in part by NSF awards CCF-0937833 and CCF-0937854, an IBM Faculty award, and a NetApp Faculty award.

Chapter 1

Introduction

Modern computer systems produce and process an overwhelming amount of data at an extremely high rates. According to International Data Corporation (IDC), the amount of stored digital information in 2011 was over 1.8 zettabytes and the number of files was over 500 quadrillion (500×10^{15}) [44]. During 2011–2016 these numbers are projected to double every year [44].

The performance of storage hardware components, however, does not keep up with the required capacity and speed at a practical cost. In fact, Hard Disk Drives (HDDs) have a low purchase price but their random access performance is often unacceptable. In addition, HDDs consume a lot of energy (6–10 watts per spindle), which increases HDDs' cost of ownership. Flash-based Solid State Drives (SSDs) have a high read performance and consume significantly less energy than HDDs but have a number of other limitations. First, though becoming less expensive, SSDs still cost significantly more than HDDs. Second, SSDs deliver significantly higher throughput than HDDs for read-intensive workloads but not for write-intensive workloads [93]. Third, wear-out effects in Flash memory necessitate a high degree of redundancy and complex firmware in SSD setups, which increases SSDs' cost. Fourth, automatic garbage collection, defragmentation, and I/O parallelization makes SSD performance unpredictable. Finally, current and theoretical peaks of memory density (in gigabytes per square inch) for SSDs is significantly lower than for HDDs. Consequently, scaling performance and capacity by purchasing more HDDs or replacing HDDs by SSDs is not a cost-effective approach.

Applying workload-based optimizations is a popular way to mitigate the widening gap between the storage performance and user requirements. Currently vendors improve throughput and latency by applying various workload-driven optimizations such as complex caching algorithms, readahead, and automatic tiering. We have little doubt that the demand for workload-aware systems will only grow.

In addition to the amount of data and its access rate, the way users access the data changes as well. The number and diversity of user applications increase. Emerging applications cause workload patterns to change rapidly and significantly. One prominent example is a shift towards virtualized environments, which causes a single storage system to serve data to many different clients (so called storage consolidation). As a result, I/O streams from multiple applications mix, perturbing I/O workload on the servers. Big data processing is another example of emerging applications that exhibit unique workload characteristics [28].

Therefore, the problem of efficient characterization of storage workloads is increasingly important. Engineers and researchers can timely design efficient storage solutions only with reliable knowledge of the present-day workloads and an accurate prediction of the future workloads. Furthermore, the problem of evaluating and comparing storage systems that incorporate workload-driven optimizations grows too; only tools and techniques that accurately preserve and synthesize realistic workload properties can evaluate the performance of such systems fairly.

Another complexity of modern storage systems comes from the high number of features that present-day users demand. Deduplication, compression, snapshotting, encryption, and other features have become almost mandatory in any modern storage array. However, these features come at a price: the performance of a feature-rich array can vary significantly depending on the features enabled and the workloads in use. For instance, a storage stack that supports deduplication can both improve and degrade system performance depending on the number of duplicate objects in the dataset. To mitigate such negative impacts, many deduplication systems implement various optimizations, e.g., a bounded hash index search against a subset of all data. In this case, performance depends on both duplicates count and spacial and temporal access locality. A proper evaluation of the trade-offs caused by different storage features requires tools that can generate workloads with realistic characteristics.

Evaluating even simple storage systems is hard. In this thesis, Chapter 2 describes the difficulties in the two accepted ways of evaluating storage systems: using synthetic benchmarks and trace replay. As the complexity of storage systems grows due to inclusion of workload-driven optimizations, the appearance of new features, and the diversification of user applications, new tools and techniques are needed for efficient workload analysis and synthesis. Chapter 3 demonstrates the basics of the Multi-Dimensional Histogram (MDH) technique by applying MDH to convert I/O traces to workload models. In Chapter 4 and Chapter 5 we apply MDH to evaluate deduplication systems and Network Attached Storage (NAS) serving the data to a set of virtual machines. We then conclude and present future work in Chapter 6. It is our thesis that MDH-based techniques are powerful for both workload analysis and synthesis. MDH bridges the widening gap between the complexity of storage systems and the availability of practical evaluation tools. The rest of this chapter summarizes the problems addressed in the thesis and describes our main contributions.

1.1 Complexities in Evaluating Storage Systems

Researchers traditionally evaluate storage systems using synthetic benchmarks and trace replay. The quality of file system benchmarking has not improved in over a decade of intense research spanning hundreds of publications. Researchers repeatedly use a wide range of poorly designed benchmarks, and in most cases, develop their own adhoc benchmarks. In addition to lax statistical rigor, the storage community lacks a definition of *what* we want to benchmark in a file system. In Chapter 2 we review a wide range of evaluation tools and techniques, and propose several dimensions of file system benchmarking. We experimentally show that even the simplest of benchmarks and conventional trace replay tools can be fragile, producing drastically different performance results even when the workloads and operating environments are almost identical. It is our hope that Chapter 2 will spur a more serious debate in the community, leading to more actions that can improve how we evaluate our file and storage systems. We argue that the MDH technique and a set of associated tools can significantly improve storage benchmarking discipline.

Tracing production systems is a traditional approach to understanding real-world workloads. The collected traces are then analyzed, relevant workload properties are extracted, and corresponding optimizations are developed. Trace *analysis* is the best option for studying representative workloads. Unfortunately, another common use of traces—*replay*—has a lot of difficulties. Traces tend to be large, hard to use and share, and inflexible in representing more than the exact system conditions at the point the traces were captured. Often, however, researchers are not interested in the precise details stored in a bulky trace, but rather in some statistical properties found in the traces—properties that affect their system's behavior under load.

Chapter 3 applies the MDH technique for converting I/O traces to workload models. We designed and built a system that (1) extracts many desired properties from a large block I/O trace, (2) builds a statistical model of the trace's salient characteristics, (3) converts the model into a concise description in the language of one or more synthetic load generators, and (4) can accurately replay the models in these load generators. Our system is modular and extensible. We experimented with several traces of varying types and sizes. Our concise models are 4–6% of the original trace size, and our modeling and replay accuracy are over 90%. We further evaluate the impact of approximations on MDH model size and accuracy.

1.2 Performance of Deduplication Systems

Historically, most I/O optimizations focused on the metadata: e.g., I/O access patterns such as random or sequential, arrival times, read/write sizes. Increasingly, storage systems also consider the *content* and not just the metadata. Deduplication is a popular component of modern storage systems, with a wide range of approaches. Unlike traditional storage systems, deduplication performance depends on the data's content as well as access patterns. Most datasets that were used to evaluate deduplication systems are either unrepresentative, or unavailable due to privacy issues, preventing an easy and fair comparison of competing algorithms. Understanding how both content and metadata evolve is critical to the realistic evaluation of deduplication systems.

In Chapter 4 we present an MDH-based model of file system changes based on properties measured on terabytes of real, diverse storage systems. Our model plugs into a generic framework for emulating file system changes. Building on observations from specific environments, our model can generate an initial file system followed by ongoing modifications that emulate the distribution of duplicates and file sizes, realistic changes to existing files, and file system growth. The framework is modular and makes it easy for other researchers to add modules specific to their environments. The models used to generate content are based on observations of many real-world datasets collected by a major storage manufacturer. In our experiments the system was able to generate a 4TB dataset within 13 hours on a machine with a single disk drive. The relative error of emulated parameters depends on the model size, but remains within 15% of real-world observations.

1.3 Virtualized Workloads

Network Attached Storage (NAS) and Virtual Machines (VMs) provide high manageability, scalability, and facilitate resource consolidation. As a result, NAS and VMs became popular in many data centers. Virtualization solutions typically encapsulate guest file systems in virtual disk images and multiplex request streams from different VMs. Consequently, NAS servers see drastically different workloads from virtualized clients than from the physical clients. Unfortunately, current

NAS workload generators and benchmarks produce workloads typical to physical machines. Consequently, the usage of current benchmarks for virtual setups requires a complex setup of hypervisors, VMs, and applications to produce realistic workloads.

Chapter 5 makes two contributions. First, we studied the extent to which virtualization is changing existing NAS workloads. We observed significant changes, including the disappearance of file system metadata operations at the NAS layer, changed I/O sizes, and increased randomness. Second, using MDH-based techniques, we created a set of versatile NAS benchmarks to synthesize virtualized workloads. New NAS benchmarks generate accurate virtualized workloads *without* the effort and limitations associated with setting up a full virtualized environment. The experiments show that the relative error of the virtualized benchmarks averages to less than 10% across 11 parameters.

Chapter 2

Complexities of Evaluating Storage Systems

In this chapter we describe the challenges of evaluating storage systems effectively. First, in Sections 2.1–2.3 we describe the problems of evaluating file systems with *synthetic benchmarks*. Then, in Sections 2.4–2.7 we cover the difficulties of using *trace replay* approach. Finally, we cover related work for this chapter in Section 2.8 and conclude in Section 2.9.

2.1 Current State of File System Performance Benchmarking

Each year, the research community publishes up to several dozens of papers proposing new or improved file and storage system solutions. Practically every such paper includes an evaluation demonstrating how good the proposed approach is on some set of benchmarks. In many cases, the benchmarks are fairly well-known and widely accepted; researchers present means, standard deviations, and other metrics to suggest some element of statistical rigor. It would seem then that the world of file system benchmarking is in good order, and we should all continue along with our current methodology.

We observe not.

We claim that file system benchmarking is actually full of incomplete and misleading results that make it virtually impossible to understand the strengths and weaknesses of a given design. In Section 2.3, we demonstrate the fragility that results when using a common file system benchmark (Filebench [39]) to answer a simple question, "How good is the random read performance of Linux file systems?" This seemingly trivial example highlights how hard it is to answer even simple questions and also how, as a community, we have come to rely on a set of common benchmarks, without really asking ourselves *what we need* to evaluate.

The fundamental problems are twofold. First, accuracy of published results is questionable even in the sciences [97] where the community welcomes the confirmation of earlier results by other researchers. Evaluating computer systems is an experimental discipline but confirming the results of existing studies is not a part of computer science culture. So, the accuracy and validity of results may be even worse in our field [119, 123]. Second, we are asking an ill-defined question when we ask, "Which file system is better." We limit our discussion here to the second point.

What does it mean for one file system to be better than another? Many might immediately focus on performance, "I want the file system that is faster!" But faster under what conditions? One system might be faster for accessing many small files, while another is faster for accessing a single large file. One system might perform better than another when the data starts on disk (e.g., its on-disk layout is superior). One system might perform better on metadata operations, while another handles data better. Given the multi-dimensional aspect of the question, we argue that the answer can *never* be a single number or the result of a single benchmark. Of course, we all know that and consequently every paper presents multiple benchmark results. But how many of those give the reader any help in interpreting the results to apply them to any question other than the narrow question being asked in that paper?

The benchmarks we choose should measure the aspect of the system on which the research in a paper focuses. That means that we need to understand precisely what information any given benchmark reveals. For example, many file system papers use a Linux kernel build as an evaluation metric [5, 75, 123, 147]. However, on practically all modern systems, a kernel build is a CPU bound process, so what does it mean to use it as a file system benchmark? The kernel build does create a large number of files, so perhaps it is a reasonable metadata benchmark? Perhaps it provides a good indication of small-file performance? But it means nothing about the effect of file system disk layout if the workload is CPU bound. The reality is that kernel compilation frequently reveals little about the performance of a file system, yet many researchers use compilation-based benchmarks nonetheless.

We claim that file systems are multi-dimensional systems, and we should evaluate them as such. File systems are a form of "middleware" because file systems have multiple storage layers above and below, and it is the interaction of all of those layers with the file system that really affects its behavior. To evaluate a file system properly we first need to agree on the different dimensions, then agree on how best to measure those different dimensions and finally agree on how to combine the results from the multiple dimensions.

In Section 2.2 we review and propose several file system evaluation criteria (i.e., a specification of the various dimensions) and then examine commonly used benchmarks relative to those dimensions. In Section 2.3 we examine 1–2 small pieces of these dimensions to demonstrate the challenges that must be addressed.

2.2 File System Dimensions

A file system abstracts some hardware device to provide a richer interface than that of reading and writing blocks. It is sometimes useful to begin with a characterization of the I/O devices on which a file system is implemented. Such benchmarks should report bandwidth and latency when reading from and writing to the disk in various-sized increments. IOmeter [100] is an example of such a benchmark; we will call these *I/O benchmarks*.

Next, we might want to evaluate the efficacy of a file system's on-disk layout. These should again evaluate read and write performance as a function of (file) size, but should also evaluate the efficacy of the on-disk metadata organization. These benchmarks can be challenging to write: applications can rarely control how a file system caches and prefetches data or metadata, yet such behavior will affect results dramatically. So, when we ask about a system's on-disk metadata layout, do we want to incorporate its strategies for prefetching? They may be tightly coupled. For example, consider a system that groups the metadata of "related files" together so that whenever you access one object, the metadata for the other objects' metadata is brought into memory. Does this reflect a good on-disk layout policy or good prefetching? Can you even distinguish them?

Does it matter? There exist several benchmarks (e.g., Filebench [39], IOzone [24]) that incorporate tests like this; we will refer to these benchmarks as *on-disk benchmarks*. Depending on how it is configured, the Bonnie and Bonnie++ benchmarking suites [21, 30] can measure either I/O or on-disk performance.

Perhaps we are concerned about the performance of metadata operations. The Postmark benchmark [68] is designed to incorporate metadata operations, but does not actually provide metadata performance in isolation; similarly, many Filebench workloads can exercise metadata operations but not in isolation.

As mentioned above, on-disk metadata benchmarks can become caching or in-memory benchmarks when file systems group metadata together. On-disk metadata benchmarks can also become in-memory benchmarks when the benchmarks sweep small file sizes or report "warm-cache" results. We claim that we are rarely interested in pure in-memory execution, which is predominantly a function of the memory system, but rather in the efficacy of a given caching *approach*; does the file system pre-fetch entire files, blocks, or large extents? How are elements evicted from the cache? To the best of our knowledge, none of the existing benchmarks consider these questions.

Finally, we may be interested in studying a file system's ability to scale with increasing load. This was the original intent behind the Andrew File System benchmark [57], and while sometimes used to that end, this benchmark, and its successor, the Linux kernel compile are more frequently cited as a good benchmark for general file system performance.

We surveyed the 2009 and 2010 publications in file systems from the USENIX FAST, OSDI, ATC, HotStorage, ACM SOSP, and IEEE MSST conferences. We recorded what benchmarks were used and what each benchmark measures. We reviewed 100 papers, 68 from 2010 and 32 from 2009, eliminating 13 papers that had no evaluation component relative to the discussion. For the rest, we counted how many papers used each benchmark. Table 2.1 shows all the benchmarks that we encountered and reports how many times each was used in each of the past two years. The table also contains similar statistics from our previous study for 1999–2007 years. We were disappointed to see how little consistency there was between papers. Adhoc testing—making one's own benchmark—was, by far, the most common choice. While several papers used micro-benchmarks for random read/write, sequential read/write and create/delete operations, the benchmarks were all custom generated. We found this surprising in light of the numerous existing tests that can generate micro-benchmark workloads.

Some of the adhoc benchmarks are the result of new functionality: three papers provided adhoc deduplication benchmarks, because no standard benchmarks exist. There were two papers on systems designed for streaming, and both of those used custom workloads. However, in other cases, it is completely unclear why researchers are developing custom benchmarks for OLTP or parallel benchmarking. Some communities are particularly enamored of trace-based evaluations (e.g., MSST). However, almost none of those traces are widely available: of the 14 "standard" traces, only 2 (the Harvard traces and the NetApp CIFS traces) are widely available. When researchers go to the effort to make traces, it would benefit the community to make them widely available by depositing them with Storage Networking Industry Association (SNIA).

In summary, there is little standardization in benchmark usage. This makes it difficult for future researchers to know what tests to run or to make comparisons between different papers.

Benchmark	**Benchmark Type**					**Used in papers**	
	I/O	**On-disk**	**Caching**	**Metadata**	**Scaling**	**1999–2007**	**2009–2010**
IOmeter [61]	●					2	3
Filebench [39]	●	○	○	○	●	3	5
IOzone [24]		○	○		●	0	4
Bonnie [21]		○	○			2	0
Postmark [68]		○	○	○	●	30	17
Linux compile		○	○	○		6	3
Compile other		○	○	○		38	14
DBench [125]		○	○	○		1	1
SPECsfs [115]		○	○	○	●	7	1
Sort [105]		○	○		●	0	5
IOR [86]		○	○		●	0	1
Production	⋆	⋆	⋆	⋆		2	2
Adhoc	⋆	⋆	⋆	⋆	⋆	237	67
Custom trace	⋆	⋆	⋆	⋆		7	18
Standard trace	⋆	⋆	⋆	⋆		14	17
BLAST [87]		○	○			0	2
FFSB [107]		○	○	○	●	0	1
Fio [40]	○	○	○		●	0	1
Andrew [57]		○	○	○		15	1

Table 2.1: Benchmarks Summary. "●" indicates the benchmark can be used for evaluating the corresponding file system dimension; "○" is the same but the benchmark does not isolate a corresponding dimension; "⋆" is used for traces and production workloads. Researchers often pick the benchmark without taking the important information above into account.

2.3 A Case Study

We performed a simple evaluation of Ext2 using Filebench 1.4.8 [39]. We picked Filebench because of its flexibility and widespread usage. E.g., in FAST 2013 six out of ten papers that performed file system benchmarking used Filebench. Nevertheless, the problems outlined by this study are common to all other benchmarks we surveyed. The range of the workloads that Filebench can generate is broad, but we deliberately chose a simple, well-defined workload: one thread randomly reading from a single file. It is remarkable that even such a simple workload can demonstrate the multi-dimensional nature of file system performance. More complex workloads and file systems will exploit even more dimensions and consequently will require more attention during evaluation. Ext2 is a relatively simple file system, compared to, say, Btrfs; more complex file systems should demonstrate more intricate performance curves along performance dimensions.

In our experiments we measured the throughput and latency of the random read operation. We used an Intel Xeon 2.8GHz machine with a single SATA Maxtor 7L250S0 disk drive as a testbed. We artificially decreased the RAM to 512MB to reduce the warmup phase of the experiments. This is a common and valid practice in the systems with Uniform Memory Access architectures. Section 2.3.1 describes our observations related to the throughput, and Section 2.3.2 highlights the latency results.

2.3.1 Throughput

In our first experiment we increased the file size from 64MB to 1024MB in steps of 64MB. For each file size we ran the benchmark 10 times. The duration of the run was 20 minutes. We ensured that it is was enough to achieve steady-state results. Figure 2.1 shows the throughput and its relative standard deviation for this experiment. The sudden drop in performance between 384MB and 448MB is readily apparent. The OS consumes some of the 512 MB of RAM and the drop in performance corresponds to the point when the file size exceeds the amount of memory available for the page cache.

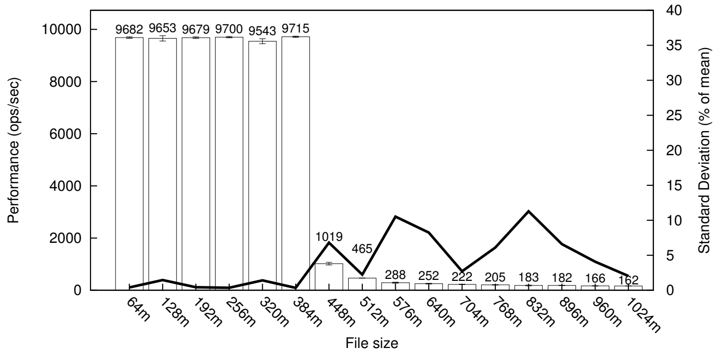


Figure 2.1: Ext2 throughput and its relative standard deviation under random read workload for various file sizes.

So, what should a careful researcher report for the random read performance of Ext2? For file sizes less than 384MB, the workload mostly exercises the memory subsystem; for file sizes greater than 448MB, the workload exercises the disk system and the page replacement algorithm. This suggests that researchers should either publish results that span a wide range of datasets or make explicit both the memory- and I/O-bound performance.

It was surprising, at first, that such a sudden performance drop happens within a narrow range of only 64MB. We ran extra experiments for file sizes between 384MB and 448MB and observed that performance drops within an even narrower region—less than 6MB in size. This happens because even a single read operation that induces I/O lasts longer than thousands of in-memory reads. This problem becomes worse in recent years as the gap between I/O and memory/CPU speeds widens. More modern file systems rely on multiple cache levels (using Flash memory or network). In this case the performance curve will have multiple distinctive steps.

Only few megabytes of RAM available for data caching can impact system performance drastically. Figure 2.1 also shows the relative standard deviation for the throughput. The standard deviation is not constant across the file sizes. In the I/O-bound range, the standard deviation is up to 5 times greater than it is in the memory-bound range. This is unsurprising given the variability of disk access times compared to the relative stability of memory performance. We observed that in the transition region, where we move from being memory-bound to being disk-bound, the relative standard deviation increases by up to 35% (not visible on the figure because it only depicts data points with a 64MB step). Just a few megabytes more (or less) available in the cache affect the throughput dramatically in this boundary region. It is difficult to control the availability of just a few megabytes from one benchmark run to another. As a result, throughput benchmarks are very fragile: just a tiny variation in the amount of available cache space can produce a large variation in performance.

We reported only the steady-state performance in the above discussion; is it correct to do so? We think not. In the next experiment we recorded the throughput of Ext2, Ext3, and XFS every 10 seconds. We used a 410MB file, because it is the largest file that fits in the page cache. Figure 2.2 depicts the results of this experiment. In the beginning of the experiment no file blocks are cached in memory. As a result all read operations go to the disk, directly limiting the throughput of all the systems to that of the disk. At the end of the experiment, the file is completely in the page cache and all the systems run at memory speed. However, the performance of these file systems differs significantly between 4 and 13 minutes.

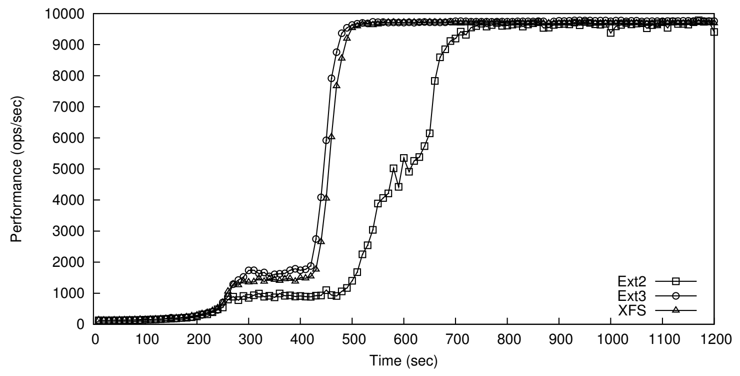
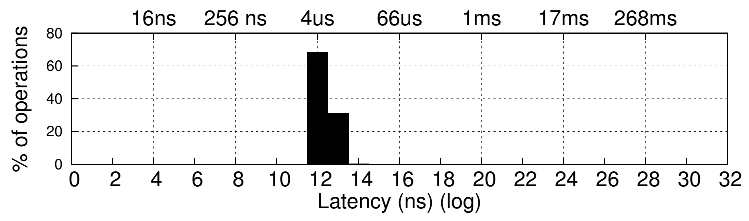


Figure 2.2: Ext2, Ext3, and XFS throughput by time. Higher value on the Y axis means better performance.

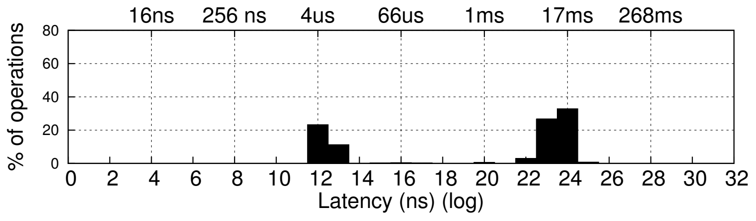
What should the careful researcher do? It is clear that the interesting region is in the transition from disk-bound to memory-bound. Reporting results at either extreme will lead to the conclusion that the systems behave identically. Depending on where in the transition range a researcher records performance, the results can show differences ranging anywhere from a few percentage points to nearly an order of magnitude! Only the *entire* graph provides a fair and accurate characterization of the file system performance across this (time) dimension. Such graphs span both memory-bound to I/O bound dimensions, as well as a cache warm-up period. Self-scaling benchmarks [27] can collect data for such graphs.

2.3.2 Latency

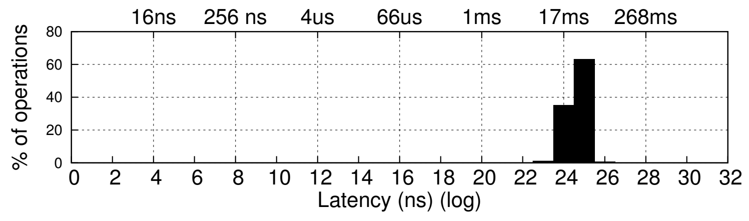
File system benchmarks, including Filebench, often report an average latency for I/O operations. However, *average* latency is not a good metric to evaluate user satisfaction when a latency-sensitive application is in question. We modified Filebench to collect latency histograms [66] for the operations it performs. We ran the same workload as described in the previous section for four different file sizes spanning a wide range: 64MB, 1024MB, and 25GB. Figure 2.3 presents the corresponding histograms. Notice that the X axes are logarithmic and that the units are in nanoseconds (above) and log_2 bucket number (below). The Y axis units are the percentage of the total number of operations performed.



(a) 64MB file



(b) 1024MB file



(c) 25GB file

Figure 2.3: Ext2 read latency histograms for various file sizes. Further to the left along the X axis is better.

For a 64MB file (Figure 2.3(a)) we see a distinctive peak around 4 *micro*seconds. The file fits completely in memory, so only inmemory operations contribute to the latency. When the file size is 1024MB we observe two peaks on the histogram (Figure 2.3(b)). The second peak on the histogram corresponds to the read calls that miss in the cache and go to disk. The peaks are almost equal in height because 1024MB is twice the size of RAM and, consequently, half of the random reads hit in the cache (left peak), while the other half go to disk (right peak). Right peak is slightly more massive in our results because 1024MB is actually more than twice of the RAM available for caching. Finally, for a file that is significantly larger than RAM—25G in our experiments—the left peak becomes invisibly small because the vast majority of the reads end up as I/O requests to the disk ((Figure 2.3(c)). Clearly, the working set size impacts reported latency significantly, spanning over 3 orders of magnitude.

In another experiment, we collected latency histograms periodically over the course of the benchmark. In this case we used a 256MB file that was located on Ext2. Figure 2.4 contains a 3D representation of the results. As the benchmark progresses, the peak corresponding to disk reads (located near the 2^{23} ns) fades away and is replaced by the peak corresponding to reads from the page cache (around 2^{11}ns). Again, depending on exactly when measurements are taken, even a careful researcher might draw any of a number of conclusions about Ext2's performance—anywhere from concluding that Ext2 is very good, to Ext2 being bad, and everywhere in between.

Worse, during most of the benchmark's run, it is bi-modal: trying to achieve stable results with small standard deviations is nearly impossible.

In summary, single number benchmarks rarely tell the complete story. We need to get away from the marketing-driven single-number mindset to a multi-dimensional continuum mindset.

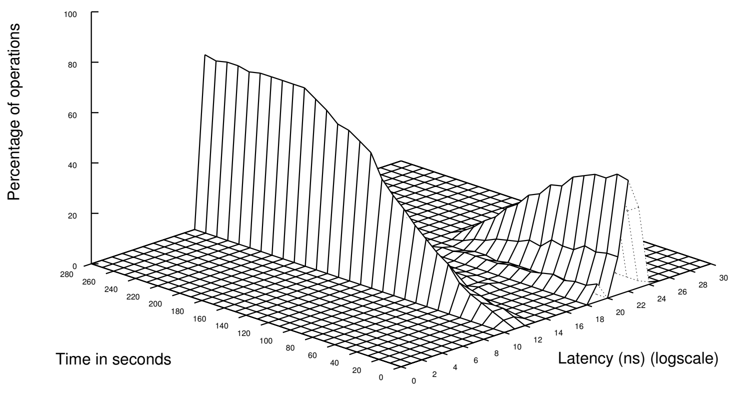


Figure 2.4: Latency histograms by time (Ext2, 256MB file). Closer to the bottom of the Latency axis is better.

2.4 Trace replay

I/O tracing is a popular tool in systems research; it is often used for workload characterization and modeling, downtime prediction, capacity planning, performance analysis, and device modeling. In this section we focus on trace replay: the re-execution of a trace to evaluate a system's performance. Trace replay is similar to an I/O benchmark, but there is a fundamental difference between running a benchmark and replaying a trace. First, researchers implicitly understand that a benchmark generates an artificial workload, even if it was designed to emulate reality [68]. In contrast, traces are usually thought to be inherently realistic, since traces often record a complex workload generated by multiple real applications and users. Therefore, the *expectation* is that the results of a trace replay will more closely represent the performance of a production system.

Second, benchmarks generally measure the *peak* performance of a system, whereas traces often capture its *typical* performance. This is because most production systems are designed to handle peak loads, and are thus underutilized most of the time [13]. A typical trace consists of 1) workload *valleys*, when utilization is low 2) workload *plateaus*, when the system is loaded, and 3) intermediate states. We use the word plateau instead of peak to emphasize that a system can be heavily loaded for extended periods of time. So if a trace is replayed by issuing the requests exactly at the times specified in the trace records (*plain replay*), peak performance will not be measured over its entire duration. Furthermore, traces are typically replayed on a newly designed and more powerful system than the one on which the traces were originally collected. In this case, plain trace replay will always keep the system underutilized. So traces need to be *scaled up*, yet there is no clear understanding of what that means. Section 2.5 discusses in detail the current approaches for replaying the traces and the issue of scaling replays. In particular, straightforward approaches to scaling result in distorted queue lengths, which can have severe performance anomalies. The two properties that differentiate trace replay from benchmarks—the expectation of accurate, realistic results and the need to scale traces—are in conflict; the results of a replay may be accurate on the original system, but by definition a scaled-up trace offers a different load than what was seen on the traced system.

Third, traces are often captured at a single layer in a system (e.g., system call, NFS, block-level, device), but the connection to the original workload is tenuous (with the possible exception

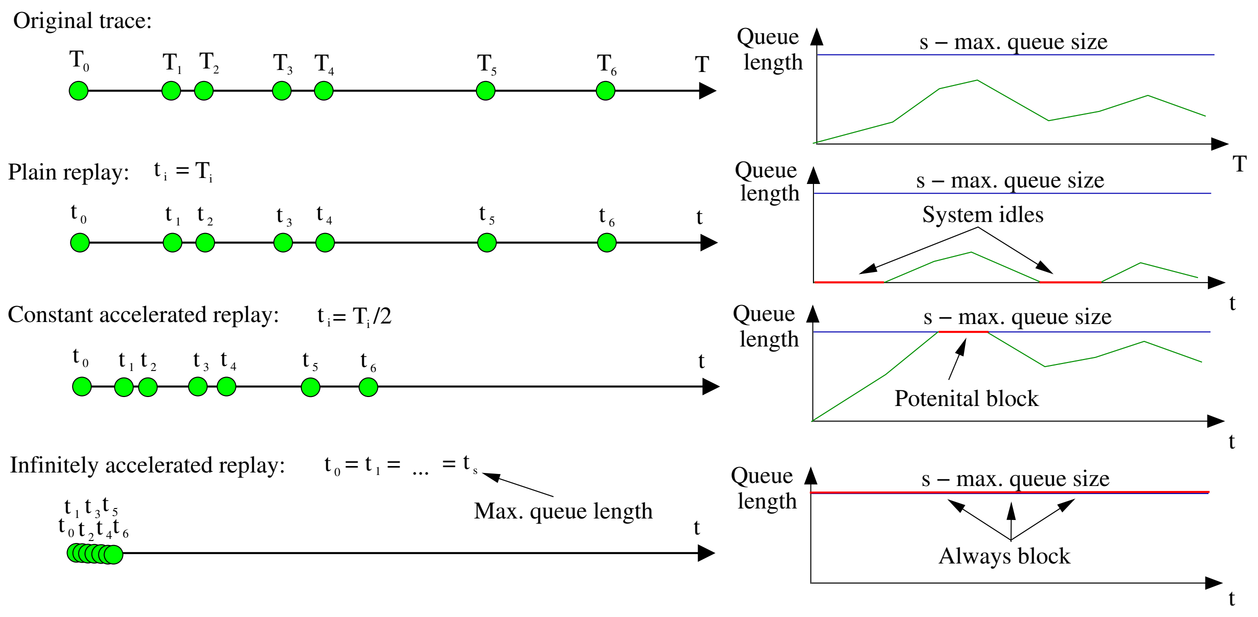


Figure 2.5: The problems of commonly used replay approaches. Capital "T" denotes a trace's original timeline, and a small "t" is a timeline during the replay on a different (presumably more powerful) system. On the right is the stylized length of an internal queue during the replay. The horizontal line "s" represents the maximum queue supported by a device.

of system-call traces). Thus, there is a tension between the need to trace at appropriate levels, which is usually driven by issues of system design and practicality, and the need to accurately record the original workload.

The fourth and final problem is that even when multiple layers are studied, most traces do not clearly identify inter-layer relationships. For example, it is not easy to distinguish whether an I/O request was caused by an incoming HTTP packet or generated due to the specifics of an implementation, e.g., periodic log file updates [29]. In fact, even if one attempts to record such relationships, it can be very difficult to extract the necessary information from the operating system's internal data structures. For example, a dirty page may have been touched by two different processes, and its eventual flush to disk may be a result of memory pressure from a third. It is unclear, when trying to characterize a workload, which of these events is the "true" cause of the disk write.

Despite these limitations, researchers often treat trace replay as a valid way to generate accurate and realistic I/O workloads. In fact, the research community believes so much in trace replays that it sometimes seems that there is a *de facto* expectation that a good paper must use trace replay in its evaluation. We show in this section that there are many limitations and complexities in the trace replay methodology.

2.5 Approaches to Trace Replay

The first challenge in generating an accurate I/O workload is to select an appropriate trace replay method that enables traces collected on one system to be representative on a completely different system. This section discusses the most common replay approaches and shows limits of existing approaches in their ability to scale the trace. We begin by discussing the three most common replay methods: *plain*, *accelerated*, and *infinitely accelerated*. We then present less popular but possibly more accurate methods.

Plain replay. The most straightforward way to do replay is to issue requests at the exact times specified in the trace. In Figure 2.5, we denote this method by $t_i = T_i$, where T_i is a relative trace time and t_i is a relative replay time of request i. Modern systems can service multiple requests in parallel by queueing them and then dispatching requests as soon as the corresponding device is free; the graph on the right side of Figure 2.5 schematically represents the length of the queue during trace collection. The horizontal line shows the maximum queue length, s. When the queue becomes full, no more requests can be accepted, and the submitter, e.g., the trace replayer or benchmark, is blocked. As we will see later, this blocking behavior is of special importance in trace replay.

Plain replay preserves idle periods and consequently is useful in evaluating certain types of power-efficient systems which often switch to low power usage modes during light loads, e.g., by spinning disks down. Sometimes plain replay also allows programmers to test systems for bugs that are triggered by some non-trivial sequence of events observed in a real system.

When plain trace replay is used on a system more powerful than the one where the trace was collected—a common occurrence—then the queue is typically shorter; it never reaches maximum size and will empty more easily, meaning that the system is idling more (Figure 2.5). Clearly, such a replay does not stress the system and cannot be used to measure peak performance. The latency distribution measured by such replay can be used to evaluate typical, but not maximum, latencies, since the internal queue is short.

Constant acceleration. One approach to scaling a trace is to issue all requests N times earlier than the times recorded in the trace (Figure 2.5). N is called the *acceleration factor* in this case. Under accelerated replay, the internal queue length for an evaluated system may be longer than for the traced system; from time to time it can reach the maximum so that the submitter is blocked, as depicted in Figure 2.5. There is no general recipe for selecting the acceleration factor. Different researchers select this parameter differently, usually without justifying the selection [149]. However, the choice of this factor can result in quite different performance numbers. More thorough evaluation of appropriate acceleration factors is beyond the scope of this thesis.

Infinite acceleration. To stress a system to its limit, the acceleration factor N can be set to infinity, as shown in the last timeline in Figure 2.5. In this case, all workload valleys are converted to plateaus, the internal queue is always full, and the replayer is always blocked. As soon as an opening appears in the queue, a request is added to it and the submitter is blocked again on submission of the next request. Keeping the queue full at all times gives the system more opportunity to perform on-line optimizations, such as request reordering and merging. This method is clearly appropriate for evaluating system's peak performance. But keeping the I/O queues full is not typical to the real systems.

Dependency-based. Often, the upper layers in real systems submit new requests only after getting the response from some previously submitted requests. Previously considered replay approaches completely neglect such dependencies. These changes in the workload can significantly skew the results of a trace-based evaluation. For example, Figure 2.6 shows the situation when the upper layers submit request R_1 only after R_0 is completed. When R_1 is finished, both R_2 and R_3 can be issued, but not R_5. If dependency information is available, accelerated replay can take it into account to improve realism of the upper layers' behavior.

The remaining question is when to issue independent requests, i.e., those that do not depend on other requests, such as R_0 and R_6 in Figure 2.6. Possible but not ideal solutions are to submit independent requests at the times specified in the trace, or as fast as possible, or with some acceleration factor.

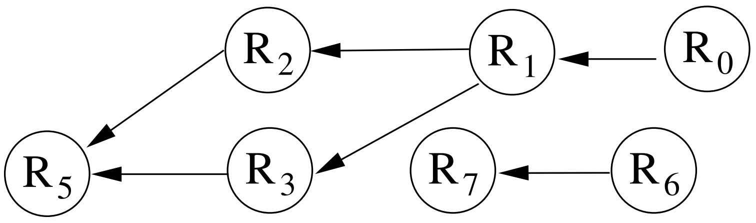


Figure 2.6: Request dependencies.

Completion-time-based. Although dependency information is missing from almost all captured traces, the request completion time is sometimes present. This gives us an opportunity to approximate dependencies based on that information.

The general idea is to submit every request as soon as possible, but only if all requests that had previously completed in the original trace have also completed during the replay. Figure 2.7 depicts an example for three requests. T_i is the recorded submission for the i-th request and T_i^c is the time it completed. When replaying, R_0 is submitted first, then R_1 immediately after that, because in the original trace R_1 was submitted before R_0 had completed. However, R_2 is not submitted until R_0 completes, just as in the original trace. Note that such an approximation can add extra dependencies that were not present in the original workload, and that the replay can be done with either constant or infinite acceleration.

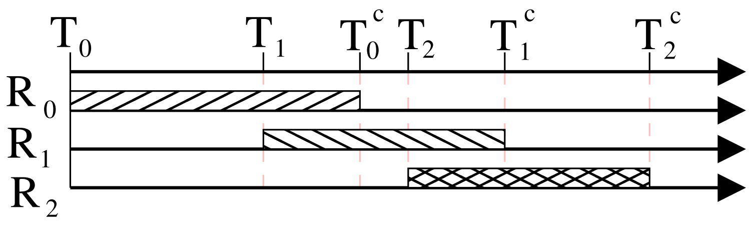


Figure 2.7: Completion-time-based replay.

2.6 Trace Replay Problems

Several fundamental issues prevent trace replay from accurately representing the original workload. They require close attention from the community before we can expect realism from a trace replay. Solving these issues would require more complete trace capture tools, better replay tools, and a much larger set of modern traces in standard formats disseminated to the community (e.g., via SNIA's growing Trace Repository [113]).

Lack of dependencies. Only a few currently available traces contain explicit information about the dependencies between requests, which drastically complicates many operations on the traces.

Some attempts have been made to extract dependencies from existing traces, but extraction is a difficult problem that can be addressed only with limited accuracy [149]. Detecting dependencies while collecting a trace may be more feasible. For example, one can put trace points at several layers in the I/O stack and detect when a single request at an upper layer induces a sequence of dependent requests at a lower one.

Think time. The time between the response from a lower layer and the submission of the next (dependent) request is called *think time*; this often corresponds to an application's computation between I/O operations, and clearly affects performance. If the completion time for each request is available in the trace, then it is possible to estimate think time. But it is unclear whether it is possible to scale the think time to other systems with different CPU and memory resources, especially when think time depends on other resources such as the network. Collecting information on the sources of think time would improve our understanding of the original workload, enabling more accurate replay. Reproducing proper think time also allows researchers to properly factor in the degradation of storage.

I/O stack. Any change in the configuration of the layers in the I/O stack influences the workload. For example, even if a file system level workload is the same, a block-level trace collected under Reiserfs is significantly different from the block-level trace collected under XFS. In fact, Reiserfs puts small files in the end of the allocated blocks (tail packing), which decreases the total number of I/O operations for many common workloads. Most traces omit information about the configuration of the I/O layers; we think that it should be a duty of tracing software to collect the layer configuration [11]. When realistic performance needs to be measured with high accuracy, then top-level traces (e.g., system calls) should be replayed.

Sometimes, the behavior of the upper layers depends on the lower layers. E.g., CFQ I/O scheduler in Linux detects flash vs. rotational block device and applies very different scheduling properties. In such cases it is questionable whether trace replay at an intermediate level is even feasible.

In addition, it is often difficult (and at times impossible) to generate intermediate-layer events with exactly the same properties without bypassing the upper layers. E.g., it is not trivial for the user-space block-level replayer to submit Linux's *bio* structures with exactly same field values as bios were submitted by a file system in the real setup. Experiment 1 in Section 2.7 demonstrates this problem in detail. A clear interface between layers, accessible to the replayer, can help mitigate this issue.

Replay duration. Many available traces cover multiple days, weeks, or even months. Most researchers do not have the luxury of replaying the trace for such a long time. On the other hand, replaying only a small sub-period of the trace might not evaluate the system properly, since it may be unrepresentative of a longer-term workload. Two possible alternatives are to select only relevant sub-periods of the trace (e.g, workload plateaus) or to randomly sample a large trace.

Workload variability. As was mentioned in Section 2.5, the workload can vary significantly within a trace: sometimes a system is highly utilized, but often it is not. Non-temporal workload properties, such as the I/O size distribution, can be quite different in plateaus and valleys.

For example, lightweight scrubbing software might run during the night and use larger I/O sizes compared to daily OLTP applications. Accelerated replay methods convert all workload valleys to plateaus, but it may not be appropriate to evaluate a system based on the valley workload. One might consider looking only at peaks if we are interested in the peak performance, but that approach might misjudge systems that favor peak workloads and exhibit bad performance for the valleys. We speculate, that if a trace is to be replayed in an accelerated fashion, its peaks and valleys should be separated and replayed separately. However, the evaluation of this conjecture is beyond the scope of this thesis.

Scaling across other parameters. Traces often need to be scaled across dimensions other than time. If a block trace contains an offset field, its value is clearly limited by the size of the traced block device. How should the offset be scaled up for use on larger disks? Should the I/O size also be scaled, or not? These questions have no easy answers, and few studies have explored spatial scaling [149].

Mmap-based accesses. Modern applications use `mmap` heavily to improve performance [51]. This can significantly distort the results of system call traces because `mmap` events show up as simple memory reads/writes. Unless the OS is modified to sample memory accesses for the purpose of tracing (e.g., using the accessed and dirty bits in the page table), only page faults can be recorded [67]. We prepared and successfully used Linux kernel patches that allow the tracer to capture mmap-based accesses. However, this is not the focus of this thesis.

2.7 Experimental Verification

To demonstrate the significance of the aforementioned problems, we designed several simple experiments that highlight them. The experiments are not meant to be complete, but rather are designed to illustrate that the community's expectations of trace replay realism are not always valid.

For our experiments we needed a *reproducible* workload generator; we used the Postmark benchmark [68] since it is widely employed by many researchers. Real workloads are more complex than what Postmark generates, so we expect our conclusions to hold even more strongly for production traces. The configuration for Postmark was selected so that we stress both the file system cache and disk I/O; it runs for at least 50 minutes on the slowest machine we tested. We used four consecutive generations of Dell servers: SC1425 (vintage 2004), SC1850 (2005), 1800 (2006), and R710 (2009). We installed the same CentOS 6.0 distribution on all machines and updated the Linux kernel to version 3.2.1. We ran Postmark and recorded a block trace on the oldest machine (SC1425). We then replayed the collected trace on all four machines using different replay approaches.

Experiment 1. The tool most commonly used for block trace replay on Linux is *btreplay*, which is part of the *blktrace* package [22]. When we used it for plain replay on the SC1425, the device I/O queue length was never as high as during the original Postmark run (Figure 2.8). As it turned out, when asynchronous I/O is used on a block device directly (the mode that is used by btreplay) all requests have the SYNC flag set. On the other hand, most requests that passed through the file system layer during the original Postmark run did not have this flag. Depending on the SYNC flag, the Linux I/O scheduler and drivers apply different policies to requests, which results in different queue lengths. We implemented a patch for the Linux kernel and btreplay to set the value of the flag as seen in the original trace and the accuracy improved significantly (see Figure 2.8). This experiment demonstrates that in many cases it is difficult to generate events with the required properties at an intermediate layer unless the upper layers are bypassed.

Figure 2.8 also shows the queue length for infinitely accelerated replay. As can be seen, the average queue length is much higher in this case, which gives a system unrealistic opportunities for on-line optimizations.

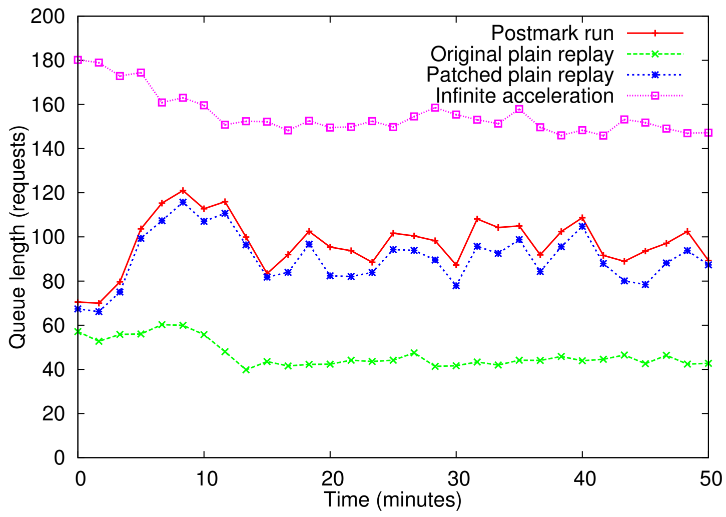


Figure 2.8: Queue length for plain replay.

Experiment 2. In the second experiment we compared the throughput reported by Postmark on each of the four machines with an infinitely accelerated replay of a trace collected on the SC1425. We expect that the more powerful a machine is, the higher are the performance numbers reported by Postmark and infinitely accelerated replay. Moreover, relative performance improvement (compared to the machine where the trace was collected) should be the same for both Postmark and trace replay. The first two columns of Table 2.2 show the throughput in operations per second for both Postmark (PM) and the replay. The next two columns present the same information, normalized to the performance of the SC1425. E.g., for SC1850, relative Postmark performance was $63/42 = 1.5$, while relative replay performance was $3,741/1,812 = 2$. The expectation is that the normalized throughput of the Postmark run should roughly match that of the replay, meaning that replay gives an accurate estimate of an application's performance. The last column in Table 2.2 shows the relative error between the normalized performances. We can see that the error is significant and is not even in a predictable direction. Interestingly, the Dell 1800 exhibited lower performance than the SC1425, which is related to the fact that write caching was disabled on the Dell 1800's controller. Because Postmark spends most of its time on I/O in this case, block-level replay accurately reflects the throughput of the application.

2.8 Related Work

Problems with synthetic I/O benchmarks. In 1994, Tang et al. criticized several file system benchmarks in widespread use at that time [119]. Surprisingly, some of these benchmarks are still in use today. In addition, plenty of new benchmarks have been developed, but quantity does not always mean quality. Traeger and Zadok examined 415 file system benchmarks from over 100 papers spanning nine years and found that in many cases benchmarks do not provide adequate

Host	**PM** (ops/sec)	**Replay** (ops/sec)	**PM** (factor)	**Replay** (factor)	**error** (%)
SC1425	42	1,812	1	1	-
SC1850	63	3,741	1.5	2	+33%
1800	16	690	0.4	0.4	<1%
R710	280	6,801	6.6	3.7	–44%

Table 2.2: Postmark (PM) vs. Replay Results.

evaluation of file system performance [123]. Table 2.1 (presented later in Section 2.2) includes results from that past study. We omit discussing those papers here again, but note that the quality of file system benchmarking does not appear to have improved since that study was published in 2008. In fact, this topic was discussed at a BoF [145] at FAST 2011, yet despite these efforts, the state of file system benchmarking remains quite poor. We believe that the main reason for that is the lack of motivation to perform quality benchmarking. In fact, to get a paper accepted it is more important to present exciting results than validate them and cover a broad spectrum of workloads and environments.

Problems of Traces. Many studies have focused on flexible trace collection with minimal interference [8, 11, 72, 91]. Other researchers have proposed trace-replaying frameworks for different layers in the I/O stack [10, 67, 149, 149]. Since a trace contains information about the workload applied to the system, a number of papers focused on trace-driven workload characterization [70, 77, 101]. N. Yadwadkar proposed to identify an application based on its trace [142]. Many studies developed workload models to generate synthetic workloads with nearly identical characteristics [18, 36, 79, 121]. The body of research related to traces is large, yet we are not aware of any papers that have questioned the realism of trace replay. There were, however, several studies that show similar problems in synthetic benchmarks [95, 120].

2.9 Conclusions

Storage stack is a complex piece of software with many layers, all affecting its overall performance. Benchmarking such systems is far more complex than any single tool, technique, or number can represent. Performing versatile and robust system evaluation makes our lives more difficult, but it will greatly enhance the utility of our work.

We believe that researchers should define precisely what dimension(s) of file system behavior researchers evaluate. We think that a file system benchmark should be a suite of nano-benchmarks where each individual test measures a particular aspect of file system performance and measures it well. Next, the community should avoid single-number reporting. Storage performance is extremely sensitive to minute changes in the environment. In the interest of full disclosure, researchers should report a range of values that span multiple dimensions (e.g., timeline, working-set size, etc.). We propose that at a minimum, an encompassing benchmark should include in-memory, disk layout, cache warm-up/eviction, and metadata operations performance evaluation components.

For many years, trace replay has been thought to be a “gold standard” in performance evaluation. However, our experiments have shown that it is difficult to replay a trace accurately even on the system where it was collected. Replaying on systems with different characteristics is sure to introduce anomalies that make performance measurements questionable at best. As a result, investigators need to be aware of these pitfalls and should select their replay techniques carefully. At the same time, further research is needed to develop methods and tools for scaling traces and ensuring the validity of replay-based conclusions.

Our community needs to buy in to doing a better job. We need to reach agreement on what dimensions to measure, how to measure them, and how to report the results of those measurements. Until we do so, our studies are destined to provide incomparable point answers to subtle and complex questions. It is our hope for this community to begin discussing these issues and eventually settle on a small subset of recommended benchmarks, tools, and techniques for future researchers to use—perhaps even form a TPC-like group.

In this thesis we propose and evaluate Multi-Dimensional Histogram technique (MDH) as a basis for benchmarking storage systems. We believe that MDH is flexible and powerful enough to be used by the whole storage community.

Chapter 3

Trace to Workload Model Conversion

3.1 Introduction

As discussed in the previous chapter, I/O tracing is an effective way to collect information about real-world workloads. The information contained in a trace allows a workload to be characterized using factors such as the exact size and offset of each I/O request, read/write ratio, ordering of requests. By replaying a trace, users can evaluate real-world system behavior, optimize a system based on that behavior, and compare the performance of different systems [69, 71, 77, 101].

Despite the benefits of traces, they are hard to use in practice. A trace collected on one system cannot easily be scaled to match the characteristics of another. It is difficult to modify traces systematically, e.g., by changing one workload parameter but leaving all others constant. Traces are hard to describe and compare in terms that are easily understood by system implementors. Large trace files can affect the system's behavior during replay by polluting the page cache or causing an I/O bottleneck [67]. Lastly, large traces are time-consuming to distribute and difficult and to work with (some modern traces are several terabytes in size [8]).

In this thesis we observe that in many cases replaying the exact trace is not required. Instead, it is often sufficient to use a synthetic workload generator that accurately reproduces certain specific properties. For example, a particular system might be more sensitive to the read-write ratio than to operation size. In this situation one does not really need to replay the trace precisely; a synthetic workload that emulates that read-write ratio would suffice. Of course, this example is simplistic, and in many cases one would be interested in more complex combinations of the workload parameters. However, the general idea that only some properties of the trace affect system behavior remains valid.

Because many systems are sensitive to only few parameters, researchers have developed many benchmarks and synthetic workload generators, such as IOzone [24], Filebench [39], and Iometer [100], which avoid many of the deficiencies of traces. But it can be difficult to configure a workload generator so that it produces a realistic workload; simple ones are not sufficiently flexible, while powerful ones like Filebench offer so many options that it can be daunting to select the correct settings.

In this work we fill the gap between traces and benchmarks by applying MDH technique to convert I/O traces into the benchmarks. We focus in this chapter on block traces because of their relative simplicity. Next two chapters extend MDH technique to file system and NFS traces.

Our system creates a universal representation of the trace, expressed as a multi-dimensional matrix in which each dimension represents the statistical distribution of a trace parameter or a function. Each parameter is chosen to represent a specific workload property. We implemented the most commonly used properties, such as I/O size, inter-arrival time, seek distance, and read-write ratio. End users can add new features as desired. Multi-dimensional histograms extracted from the traces allow researchers to understand the workloads quickly and facilitate the scaling of the workloads across parameters.

For each benchmark, a small plugin converts the universal trace matrix into the specific benchmark's language. To create an accurate plugin for a specific benchmark the researcher should be an expert in the benchmark's language. In this chapter, we present an implementation of Filebench plugin and evaluate its accuracy.

Many workloads vary significantly during the tracing period. To address this issue, our system supports trace *chunking* across time. Within each chunk, we consider the workload to be stable and uniform and express it as a separate matrix. For chunks with similar matrices (and consequently similar workloads) we store only one matrix. This significantly reduce the size of the model for workloads with a high degree self-similarity [47].

We evaluated the accuracy of our system by generating models from several publicly available traces. We first replayed each trace on a test system, observing throughput; latency; I/O queue length and utilization; power consumption; request sizes; CPU and memory usage; and the numbers of interrupts and context switches. Then we emulated the trace by running benchmarks on the same system, collected the same measurements, and compared the results. We computed the error of workload emulation as a Root Mean Square (RMS) and maximum distances between the measurements.

Our RMS error was less than 10% on average, and the maximum error was 15% across all experiments. Users can control the error by varying several parameters. E.g., the smaller is the initial chunk size and the lower is the matrix similarity threshold the lower is the error of a model. The speed of conversion depends on the number of dimensions in the matrix, the chunking algorithm, and the complexity of the benchmark plugin. For a basic set of metrics, we converted a 1.4GB trace to the Filebench language in only 30s. The resulting trace description was 60MB, or 23.3× smaller.

Notice that traces are still extremely valuable to detect and extract properties. We are not trying to replace the traces, we are adding a powerful tool to the storage evaluation toolbox. Conversion of I/O traces to benchmarks solves several problems highlighted in Chapter 2. First, resulting benchmarks are provably representative of a specific workload in the original trace. Second, the model allows the researcher to quickly grasp which mode of the system does the workload stress: e.g., sequential or random throughput, or in-RAM or out-of-RAM performance. Finally, modification of a histogram (e.g., for workload scaling or testing what-if scenarios) is much easier operation than modifying large trace files.

3.2 Design

Our five design goals, in decreasing priority, are:

1. **Accuracy**: Ensure that trace replay and trace emulation yield matching evaluation results.

2. **Flexibility**: First, leverage existing powerful workload generators, rather than creating new ones. Therefore, traces should be translated into models that can be accurately described using the capabilities of existing benchmarks. Second, allow users to choose anything from accurate yet bulky models to smaller but less precise ones.

3. **Extensibility**: Allow the model to include additional properties chosen by the user.

4. **Conciseness**: The resulting model should be much smaller than the original trace. An outcome of this is less cumbersome storage and transferring of traces, elimination of bottlenecks during replay. It will also make easier to compare and understand workloads in traces.

5. **Speed**: The time to translate large traces should be reasonable even on a modest machine.

Feature Extraction. The first step in our model-building process is to extract important features from the trace. We first discuss how we extract parameters from workloads whose statistical characteristics do not change over time, i.e., stationary workloads. Then we describe how to emulate a non-stationary workload.

Each block trace record has a set of fields to describe the parameters of a given request. Fields may include the operation type, offset or block number, I/O size, timestamp. Our translator is field-oblivious: it considers every parameter as a number. We designate these parameters as an n-dimensional vector $\vec{p} = (p_1, p_2, ..., p_n)$.

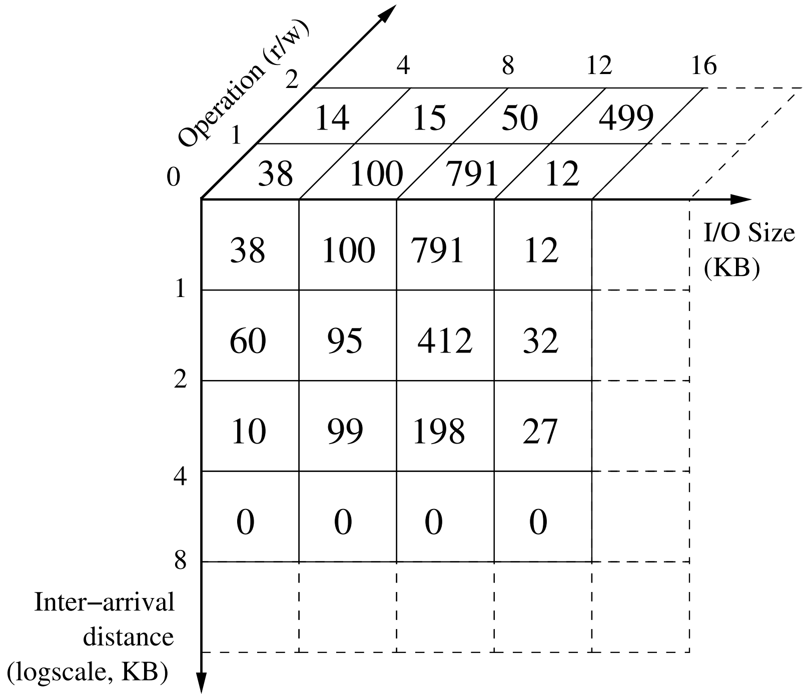


Figure 3.1: Workload representation using a feature matrix.

We define a *feature function* vector on $\vec{p}$:

$$\vec{f} = (f_1(\vec{p}, s_1), f_2(\vec{p}, s_2), ..., f_m(\vec{p}, s_m)) = \vec{f}(\vec{p}, s_f)$$

Each feature function represents an analysis of some property of the trace; s_i represents private state data for the i-th feature function, which lets us define features across multiple trace entries and parameters. Stateless feature functions ignore s_i.

For example, assume that p_1 and p_2 represent the I/O size and offset fields, respectively. We can then define the simple feature functions f_1—just the I/O size itself—and f_2—the logarithmic inter-arrival distance (offset difference between two consecutive requests):

$$f_1 = f_1(\vec{p}, s_1) = p_1$$

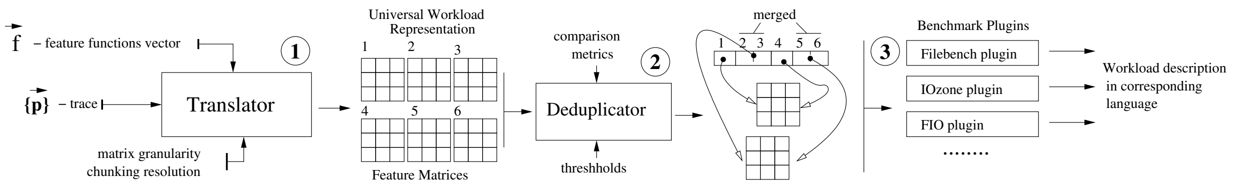


Figure 3.2: Overall trace-to-model system design.

$$f_2 = f_2(\vec{p}, s_2) = \log(p_2 - s_2.\mathit{prev\_offset})$$

In our translator, the user first chooses a set of m feature functions. Evaluating these functions on a single trace record results in a vector that represents a point in an m-dimensional feature space. The translator divides the feature space into buckets of user-specified size, and collects a histogram of feature occurrences in a multi-dimensional matrix—the *feature matrix*—that explicitly captures the relevant statistics of the workload, and implicitly records their correlations.

For example, using the two feature functions above, plus a third that encodes the operation (0 for reads, 1 for writes), the resulting feature matrix might look like the one in Figure 3.1. In this case, the trace held 52 requests of size less than 4KB and inter-arrival distance less than 1KB; of those, 38 were reads and 14 were writes.

By choosing a set of feature functions, users can adjust the workload representation to capture any important trace features. By selecting an appropriate bucket granularity, users can control the accuracy of the representation, trading off precision for computational complexity in the translator and matrix size. Stage 1 in Figure 3.2 shows the translator's role in the overall design.

Once the feature matrix has been created, the translator can perform a number of additional operations on it: projection, summation along dimensions, computation of conditional probabilities, and normalization. These operations can be used by the benchmark plugins (described below) to calculate parameters. For example, using the matrix in Figure 3.1, a plugin might first sum across the distance-vs.-size plane to calculate the total numbers of reads and writes, normalize these to find P(read), and then generate benchmark code to conditionalize I/O size on the operation type.

Clearly, the choice of feature functions affects the quality of the emulation; currently the investigator must do this based on the insight into the particular system of interest, e.g., whether it has been optimized for certain workloads that can be reflected in an appropriate feature function. We have implemented a library of over a dozen standard feature functions based on those commonly found in the literature [36, 38, 78, 84], including operation type, I/O size, offset distribution, inter-arrival distance, inter-arrival time, process identifier, etc. New feature functions can easily be added as needed to capture specialized system characteristics.

Benchmark Plugins. Once a feature matrix has been constructed from trace, it is possible to use it directly as input to a workload generator. However, our goal in this thesis is not to create yet another generator. Instead, we believe that it is best to build on the work of others by using existing workload generators and benchmarks. This approach allows us to easily reuse all the extensive facilities that these benchmarks provide. Many existing benchmarks offer a way to configure the generated workload; some offer command-line configuration parameters (e.g., IOzone [24] and Iometer [100]) while others offer a more extensive language for that purpose (e.g., Filebench [39]).

Most existing benchmarks use statistical models to generate a workload. Some of them use average parameter values; others use more complex distributions. In all cases, our feature matrices contain all the information needed to control the models used by these benchmarks. A simple plugin translates the feature matrix into a specific benchmark's parameters or language. For some benchmarks, the expressiveness of the parameters might limit the achievable accuracy, but even then the plugin will help choose the best settings to emulate the original trace's workload. Stage 3 in Figure 3.2 demonstrates the role of the benchmark plugins in the overall design.

Our t2m converter is flexible enough to perform conversion to any language. For our investigations, we have implemented plugins for Filebench and IOzone. We chose Filebench for its flexibility, and IOzone because it is more suitable for micro-benchmarking. We found that it was easy to add a plugin for a new benchmark, since only a single function has to be registered with the translator. The size of the function depends on the number of feature functions and the complexity of the target benchmark. For feature functions in our library, the plugins never exceeded 300 lines of code.

Chunking. Many real-world traces are non-stationary: their statistical characteristics vary over time. This is especially true for traces that cover several hours, days, or weeks. However, most workload generators apply a stationary load, and cannot vary it over time. We address this issue with *trace chunking*: splitting a trace into chunks by time, such that the statistics of any given chunk are relatively stable. Finding chunk boundaries is difficult, so we first use a constant user-defined chunk size, measured in seconds. For each chunk, we compute a feature matrix independently; this results in a sequence of matrices. We then convert these fixed chunks into variable-sized ones by feeding the matrices to a deduplicator that merges adjacent similar matrices (Stage 2 in Figure 3.2). This optimization works well because many traces remain stable for extended periods before shifting to a different workload mode. We normalize the matrices before comparing them, so that the absolute number of requests in a chunk does not affect the comparison. We use the maximum distance between matrix cells as a metric of similarity. When two matrices are found to be similar, we average their values and use the result to represent the workloads in the corresponding time chunks.

Besides detecting varying workload phases, the deduplication process also reduces the model size. To achieve even further compression, we support all-ways deduplication: every chunk in a trace is deduplicated against every other chunk (not just adjacent ones).

Along with the matrices, we generate a time-to-matrices map that serves as an additional input to the benchmark plugins. If the target benchmark is unable to support a multi-phase workload, the plugin generates multiple invocations with appropriate parameters.

In the example in Figure 3.2, we set the trace duration to 60s and the initial chunk size to 10s, so the translator generated six matrices. After all-ways deduplication, only two remained.

3.3 Mathematical Approximations

Multi-Dimensional Histogram technique (MDH) is based on empirical distributions collected from real traces. In other words, MDH collects the absolute or relative numbers of trace features in the appropriate histogram buckets. As a result, MDH needs to maintain the information about every non-empty bucket, which sometimes makes the size of the model large. Large models are hard to understand and analyze, occupy a lot of space, and are more difficult to replay without impacting the system under evaluation. In this section, we describe how empirical distributions can potentially be approximated with mathematical functions to reduce the size of the model [124].

Figure 3.3 demonstrates an example of such approximation for a single-dimensional histogram. Instead of storing the value of every point in a histogram, the formula can be defined using a limited set of parameters that characterize the complete distribution. The usage of mathematical functions decreases the size of the model and allows us to describe workloads in a concise way. Also, formulas, being continuous mathematical objects can be processed using powerful calculus methods, e.g., differentiation and integration. In the future, these methods can help to obtain new workload characteristics that were harder to identify earlier using the discrete methods.

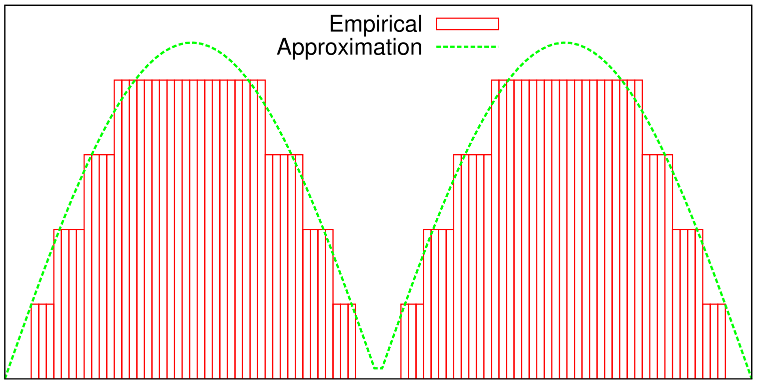


Figure 3.3: Approximation of an empirical distribution.

The downside of using mathematical functions instead of empirical distributions is decreased accuracy of the workload model. We evaluate the impact of approximation on model accuracy and size in Section 3.5.1. In the following sections we describe the details of our approximation procedure.

3.3.1 Filebench Custom Variables

To synthesize workloads using mathematical functions, corresponding workload generator should be able to generate random values with the required mathematical distributions. We used Filebench as a workload generator. Almost any parameter in Filebench's workload description (e.g., I/O size) can be assigned to a random variable. Doing that instructs Filebench to generate a new parameter value every time the parameter is used. Currently Filebench supports only empirical, uniform, and gamma random variables. In our experience, real-world distributions are diverse and complex, so we needed to extend the list of distributions supported by Filebench. Since we do not know a priori which functions approximate real-world empirical distributions the best, we need a convenient way for adding new distributions. We enhanced Filebench by adding a support for arbitrary mathematical distributions via the new type of variables—*custom variables*. We use the term *custom* instead of *random*, because custom variables, unlike random variables, might not be stationary. I.e., custom variable distribution might change over time. In future, this property can be used to avoid trace chunking and approximate temporal changes in a workload using mathematical functions.

To achieve high extensibility we added the notion of *custom variable plugins* to Filebench. Each plugin implements a single distribution for a custom variable. Plugins are distributed as dynamically linked objects that implement a simple API described below. In the beginning of execution, Filebench looks for plugins in a predefined directory and loads all of them in memory.

Workload model description can then define custom variables of corresponding type and assign parameters to custom variables.

Each plugin implements a simple API required by Filebench. The two main functions are `cvar_alloc_handle()` and `cvar_next_value()`. Filebench calls the former one upon the definition of a new custom variable, and the latter one every time a new value of the variable is needed. A user can easily implement libraries with desired custom variable behavior and immediately use it in Filebench's workload description. To the best of our knowledge, there is no other benchmark with such a high degree of flexibility.

3.3.2 Mersenne Twister Pseudo Random Number Generator

Using the newly designed custom variable API we implemented nine plugins that provide the following statistical distributions: 1) empirical; 2) exponential; 3) erlang (or gamma); 4) normal; 5) lognormal; 6) polynomial; 7) triangular; 8) uniform, and 9) weibull.

Our plugins use *mtwist* library—Geoff Kuenning's implementation of Mersenne Twister Pseudo Random Number Generator (PRNG) algorithm [76]. First, mtwist already supports all of the distributions listed above (except the polynomial one). Second, Mersenne Twister algorithm and mtwist implementation specifically is significantly faster than other PRNGs. In our experiments, Linux's `/dev/urandom` file was able to generate 1.25 million 64-bit random numbers per second, libc's `random()` function about 145 million numbers per second, and mtwist about 170 million numbers per second. Having a fast random number generator is crucial when evaluating high-speed storage. In fact, generator's execution time can contribute a lot to the relatively short I/O request latencies of fast storage. In addition to being fast, the quality of random numbers generated by Mersenne Twister algorithm is known to be very high [89].

3.3.3 Approximation Algorithm

MDH technique initially converts an I/O trace to empirical distribution. Next, we approximate the distribution by a mathematical formula using Levenberg–Marquardt Algorithm (LMA) [88]. Given a target parameterized formula, LMA finds the parameters' values that minimize the least square distance between the empirical distribution and the formula. Target formula can be any function e.g., polynomial, trigonometric, or exponential one.

According to the Teylor's theorem, any function can be approximated with a required accuracy by a polynomial of an appropriate length [53]. As a result, polynomial approximation is a widely used and accepted way to approximate functions [124]. In this study we used GNU Scientific Library to approximate empirical distributions with polynomial curves [45].

Our approximation engine takes two parameters as an input: 1) empirical MDH; and 2) target approximation error. The output of the engine is the smallest polynomial P that approximates given MDH with the target approximation error. If the amount of space for storing P is larger than the size of original MDH than the approximation is not effective for reducing the model size. However, if the size of P is smaller than the model size of MDH then the model size reduces. Section 3.5.1 evaluates the impact of this approximation procedure on model size and accuracy.

Characteristic	**Finance1**	**MS-WBS**
Duration	12 hours	1.5 hours
Reads/Writes (10^6)	1.2/4.1	0.7/0.6
Avg I/O size	3.5KB	20KB
Seq. Requests	11 %	47%

Table 3.1: High-level characteristics of the used traces.

3.4 Implementation

Traces from different sources often have different formats. We wanted our translator to be efficient and portable. We chose the efficient and flexible DataSeries format [9]—recommended by the Storage Networking Industry Association (SNIA)—and we selected SNIA's draft block-trace semantics [114]. We wrote converters to allow experimentation with existing traces in other formats. We also created a block-trace replayer for DataSeries, which supports several commonly used replay modes. In total we wrote about 3,700 LoC: 1,500 in the translator, 800 in the converters, 1,000 in the DataSeries replayer, and 400 in the Filebench and IOzone plugins.

3.5 Evaluation

System Response. To evaluate a system empirically researchers measure the system's response to an applied workload. Performance is often characterized by throughput, latency, CPU utilization, I/O queue length, and memory usage [123, 140]. Power consumption characterizes energy efficiency [83, 110], and the wear or error rates of flash devices can be tracked to evaluate reliability [32, 102].

If all response metrics are similar between trace replay and trace modeling then the trace is modeled properly. To evaluate the accuracy of our trace extraction and modeling system, we surveyed papers in Usenix FAST conferences from 2008–2011 and noted that the frequently used metrics fell into four categories: (1) throughput and latency; (2) I/O utilization and average I/O queue length; (3) CPU utilization and memory usage; and (4) power consumption. Most of the surveyed papers included 1–2 of these metrics, but in our study we evaluate all four types to ensure a comprehensive comparison. Our system is modular and easily extensible to emulate any additional metrics one desires.

During all runs we collected the accuracy parameters specified above using the *iostat*, *vmstat*, and *wattsup* tools; we plotted graphs showing the value of each accuracy parameter versus time for both replay and emulation. We include the graphs for several representative accuracy parameters and average and maximum emulation error for all parameters.

To evaluate the accuracy, conversion speed, and compression of our system, we used multiple micro-benchmarks and a variety of real traces. We present evaluation results based on two traces in this thesis: Finance1 [126] and MS-WBS [70]. The Finance1 trace captures the activity of several OLTP applications running at two large financial institutions. The MS-WBS traces were collected from daily builds of the Microsoft Windows Server operating system. The high-level characteristics of the traces are presented in Table 3.1.

It is fair to assume that the accuracy of our translator might depend on the system under evaluation. In our experiments we used a spectrum of block devices: various disk drives, flash drives, RAIDs, and even virtual block devices. We present results from two extremes of the spectrum. In the first experimental setup—*Setup P*—we used a *P*hysical machine with an external SCSI Seagate Cheetah 300GB disk drive connected through an Adaptec 39320 controller. The fact that the drive was powered externally allowed us to measure its power consumption using a WattsUp meter [136].

The second experimental setup (*Setup V*) is an enterprise-class system that has a *V*irtual machine running under the VMware ESX 4.1 Hypervisor. The VM accesses its virtual disks on an NFS server backed by a GPFS parallel file system [58, 109]. The VM runs CentOS 6.0; the ESX and GPFS servers are IBM System x3650's, with GPFS using a DS4700 storage controller. Accuracy metrics were recorded at the NFS/GPFS server.

On both setups, we first replayed traces and then emulated them using Filebench. In all experiments we set the chunk size to 20s and enabled all feature functions. We chose the matrix granularity for each dimension experimentally, by decreasing it until the accuracy began to drop.

Figure 3.4 depicts how the throughput for both reads and writes changes with time for the Finance1 trace. The replay was performed with infinite acceleration, as if we evaluate system's peak performance; it took about 5 hours to complete on Setup P. The trace emulation line closely follows the replay line; the Root Mean Square (RMS) distance is lower than 6% and the maximum distance is below 15%.

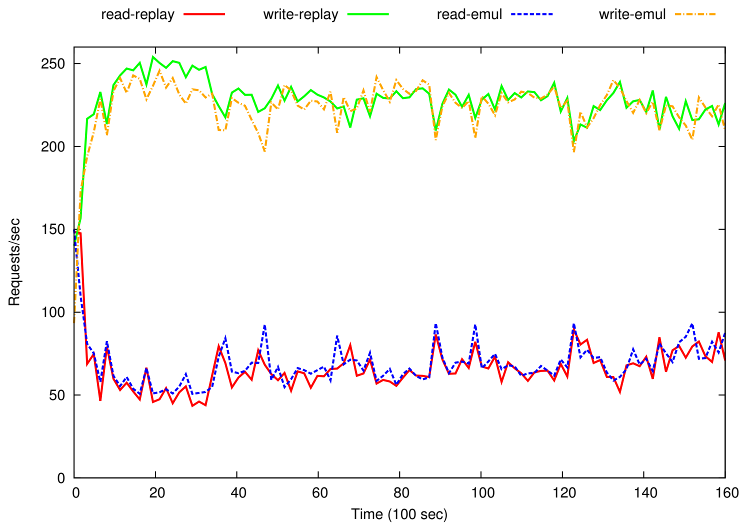


Figure 3.4: Reads and writes per second, Setup P, Fin1 trace.

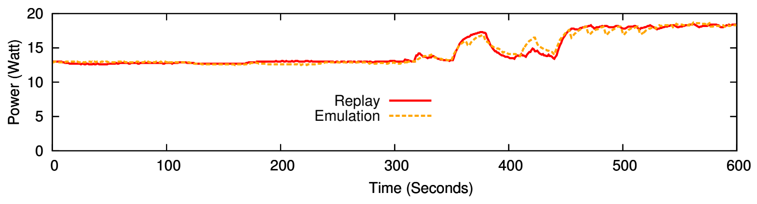


Figure 3.5: Disk power consumption, Setup P, MS-WBS trace.

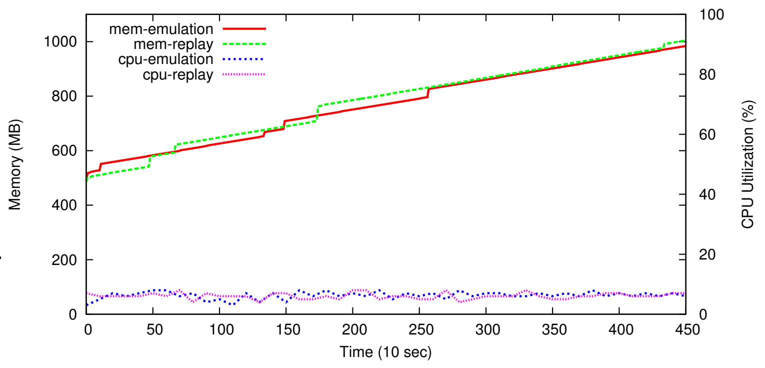
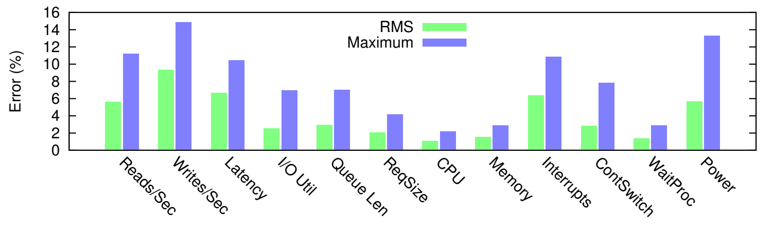
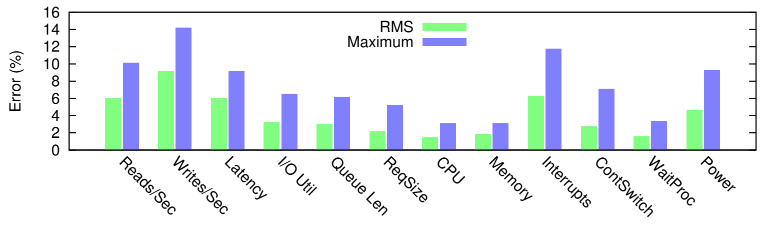


Figure 3.6: Memory and CPU usage, Setup P, Fin1 trace.

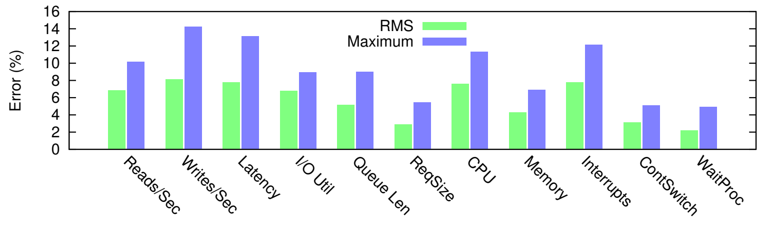
In the beginning of the run, read throughput was 4 times higher then later in the trace. By inspecting the model we found that the workload exhibits high sequentiality in the beginning of the trace. After startup, the read throughput falls to 50–100 ops/s, which is reasonable for an OLTP-like workload and our hardware. The write performance is 2–2.5 times higher than for read, due to the controller's write-back cache that makes writes more sequential.



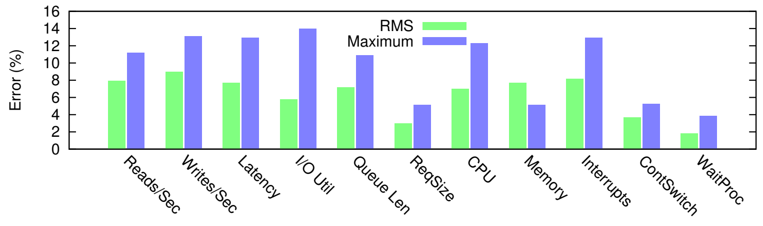
(a) Setup P, Fin1 trace



(b) Setup P, MS-WBS trace



(c) Setup V, Fin1 trace



(d) Setup V, MS-WBS trace

Figure 3.7: Root Mean Square (RMS) and maximum relative distances of accuracy parameters for two traces and two systems.

Figure 3.5 depicts disk-drive power consumption in Setup P during a 10-minute non-accelerated replay and emulation of the MS-WBS trace. In the first 5 minutes trace activity was low, resulting in low power usage. Later, a burst of random disk requests increased power consumption by almost 40%. The emulation line deviates from the replay line by an average of 6%.

In Setup V, the GPFS server was caching requests coming from a virtual machine. As a result, the run time of the Fin1 trace was only 75 minutes. The memory and CPU consumption of the GPFS server during this time are shown in Figure 3.6. Memory usage rises steadily, increasing by about 500MB by the end of the run, which is the working-set size of the Fin1 trace. Discrepancies between replay and emulation are within 10%, but there are visible deviations at times when the memory usage steps up. We attribute this to the complexity of the GPFS's cache policy, which is affected by a workload parameter that we did not emulate. CPU utilization remained steadily about 10% for both replay and emulation.

Figure 3.7 summarizes the errors for all parameters, for both setups and traces. The maximum emulation error was below 15% and RMS distance was 10% on average. Although the maximum discrepancy might seem high, Figure 3.4 shows sufficient behavioral accuracy.

The selection of feature matrix dimensions is vital for achieving high accuracy. If a system is sensitive to a workload property that is missing in the feature matrix, accuracy can suffer. For example, disk- and SSD-based storage systems may have radically different queuing and prefetching policies. To ensure high-fidelity replays across both types of systems, the feature matrix should capture the impact of appropriate parameters.

The chunk size and matrix granularity also affect the model's accuracy. Our general strategy is to select these parameters liberally at first (e.g., 100s chunk size and 1MB granularity for I/O size) and then gradually and repeatedly restrict them (e.g., 10s chunk size, 1KB I/O size) as needed until the desired accuracy is achieved. One can always be guaranteed to get high enough accuracy if sufficiently small numbers are used.

Conversion Speed and Model Size. The speed of conversion and the size of the resulting model depend on the trace length and the translator parameters. On our 2.5GHz server, traces were converted at about 50MB/s, which is close to the throughput of the 7200RPM disk drive. The resulting model without deduplication was of approximately 10–15% size of the original trace. Deduplication removed over 60% of the chunks in both the Fin1 and MS-WBS traces, resulting in a final model size reduction of 94–96%. All sizes were measured after compressing both traces and models using bzip2.

In addition to the general purpose bzip2 compressor, we used Tcgen 2.0—a tool to automatically generate lossless trace compressors [23]. This tool generates a compressor given a user specified trace format. The compressor then uses a set of predictors, similar to the ones used in CPU branching engines, to compress the trace. Tcgen compressed our traces 5–6 times better than bzip2. This is a significant improvement for lossless compression. Notice, that Tcgen was originally developed to compress CPU traces. We believe that in future it can be optimized for compressing I/O traces but this is beyond the scope of this thesis. The trace-to-model conversion technique proposed in this thesis achieves 3–5$\times$ higher compression ratio, allows to trade model accuracy for its size, and generates the models suitable for workload analysis and processing.

The size of an MDH model depends on the number of features that a user defines. Moreover, in the worst case adding each new dimension can multiply the size of the matrix. However, for the practical number of dimensions and real workloads, the size of the model was always significantly smaller than the trace size. The reason for that is twofold. First, the matrix is sparse and as a result adding a new dimension does not cause the number of points in the histogram to multiply. Second, deduplication removes over 60% of the duplicated chunks because traces exhibit a lot of self-similarity [47].

3.5.1 Approximation

In this section we evaluate how various approximations impact trace-to-model conversion accuracy and the resulting model size.

Converter parameters. Our trace-to-model converter takes several parameters as an input: initial chunk size, similarity metric and threshold, matrix granularity. The qualitative impact of these parameters on the system is intuitively clear. The smaller is the initial chunk size, the higher is the accuracy of the model and the larger is the model size. The lower is the similarity threshold, the fewer chunks are deduplicated and the larger is the model. The smaller is matrix granularity, the higher is the accuracy of the model. We quantitatively evaluated the initial chunk size impact on the model accuracy.

Figure 3.8 depicts the dependency between the chunk size and the resulting accuracy. Chunk size 1 corresponds to the case when only single request resides in a chunk. Consequently, the model error is low—below 6%. As the chunk size increases the error grows. Interestingly, chunks of size 80 seconds show similar accuracy to the 20 second chunks. It means that sometimes larger chunks can be used to reduce the size of a model without jeopardizing the accuracy. In our experiments, 80 second chunks were producing models of about 1.5× smaller than the 20 second chunks.

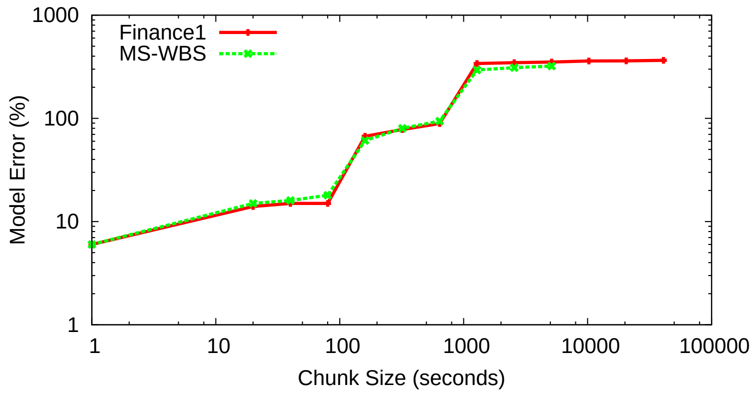


Figure 3.8: Model accuracy depending on chunk size.

As the chunk size increases beyond 80 seconds, however, the error increases significantly—up to 336% when the whole trace is represented by a single chunk (the right most point on the graph). Notice, that for both Finance1 and MS-WBS traces the graphs are similar and therefore identical initial chunking size can be used.

Mathematical Approximations. To evaluate how helpful is polynomial approximation for reducing the model size we designed the following experiment. We were varying the approximation target error from 0 (no approximation, empirical distribution used for all chunks) to 100% target error. For every target error we measured the size of a model and its accuracy. Figure 3.9 presents the results of this experiment. Between 0% and 10% target error the model's size and error remain the same, indicating that no polynomials smaller than empirical distribution can be found. As the target error is relaxed beyond 10%, the model size starts to slowly decrease, getting to 88% in the end of the graph. However, the model error grows much faster (almost linearly) with the target error.

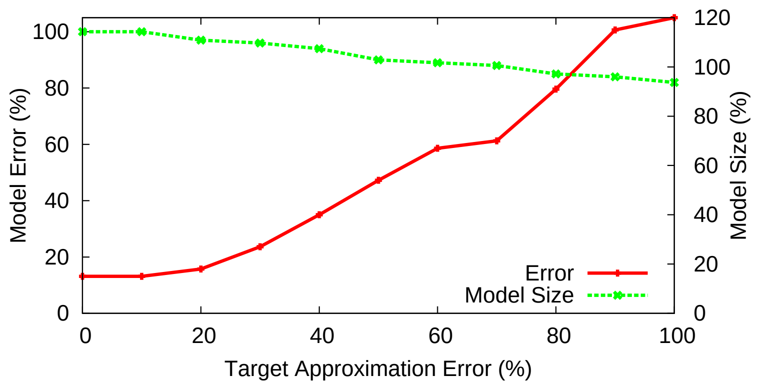


Figure 3.9: Model size (green line, right Y axis) and error (red line, right Y axis) depending on the target error. 0% target error corresponds to no approximation.

This experiment demonstrates that real-world distributions are more complex than what polynomial functions can represent. In future, other target functions can be experimented with. All required infrastructure for this, as explained in Section 3.3.3, is in place.

3.6 Related Work

Many studies have focused on accurate trace collection with minimum interference [8, 11, 72, 91, 96]. Other researchers have proposed trace-replaying frameworks at different layers in the storage stack [10, 67, 149, 149, 150]. Since a trace contains information about the workload applied to the system, a number of works focused on trace-driven workload characterization [70, 71, 77, 101]. N. Yadwadkar proposed to identify an application based on its trace [142].

After a workload is characterized, a few researchers have suggested a workload model that allows them to generate synthetic workloads with identical characteristics [18, 41, 42, 46, 55, 56, 134, 135, 146]. These works address only one or two workload properties, whereas we present a general framework for any number of properties. Also, we chunk data and generate workload expressions for the languages of already existing benchmarks.

The two projects most closely related to ours are Distiller [79] and Chen's Workload Analyzer [29]. Distiller's main goal is to identify important workload properties. We can use this information to properly define dimensions for our feature matrix. Distiller uses only latency as an accuracy parameter, focuses on disk-array-based systems, does not perform chunking, and implements only one workload generator. Chen uses machine learning techniques to identify the dependencies between workload features. However, the authors do not emulate traces based on the extracted information.

3.7 Conclusions

We have created a system that extracts flexible workload models from large I/O traces. Through the novel use of chunking, we support traces with time-varying statistical properties. In addition, trace extraction is tunable, allowing model accuracy and size to be traded off against creation time. Existing I/O benchmarks can readily use the generated model by implementing a plugin. Our evaluation with Filebench and several block traces demonstrated that the accuracy of generated models approaches 95%, while the model size is less than 6% of the original trace size. Such concise models allow easy comparison, scaling and other modifications.

Chapter 4

Realistic Dataset Generation

4.1 Introduction

The amount of data that enterprises need to store increases faster than prices drop, causing businesses to spend more on storage [19]. One way to reduce costs is deduplication, in which repeated data is replaced by references to a unique copy; this approach is effective in cases where data is highly redundant [65, 92, 103]. For example, typical backups contain copies of the same files captured at different times, resulting in deduplication ratios as high as 95% [48]. Likewise, virtualized environments often store similar virtual machines [65]. Deduplication can be useful even in primary storage [92], because users often share similar data such as common project files or recordings of popular songs.

The significant space savings offered by deduplication have made it an almost mandatory part of the modern enterprise storage stack [31, 98]. But there are many variations in how deduplication is implemented and which optimizations are applied. Because of this variety and the large number of recently published papers in the area, it is important for community to accurately compare the performance of deduplication systems.

The standard approach to deduplication is to divide the data into *chunks*, hash them, and look up the result in an index. Hashing is straightforward; chunking is well understood but sensitive to parameter settings. The indexing step is the most challenging because of the immense number of chunks found in real systems.

Three primary evaluation criteria for deduplication systems are (1) space savings, (2) performance (throughput and latency), and (3) resource usage (disk, CPU, and memory). All three metrics are affected by the data used for the evaluation and the specific hardware configuration. Although previous storage systems could be evaluated based only on the I/O operations issued, deduplication systems need the actual content (or a realistic re-creation) to exercise caching and index structures.

Datasets used in deduplication research can be roughly classified into two categories. (1) Real data from customers or users, which has the advantage of representing actual workloads [33, 92]. However, most such data is restricted and has not been released for comparative studies. (2) Data derived from publicly available releases of software sources or binaries [62, 141]. But such data cannot be considered as representative of the general user population. As a result, neither academia nor industry have wide access to representative datasets for unbiased comparison of dedup systems.

Using MDH technique we created a framework for *controllable data generation*, suitable for evaluating deduplication systems. Our dataset generator operates at the file-system level, a common denominator in most deduplication systems: even block- and network-level deduplicators often process file-system data. Our generator produces an initial file system image or uses an existing file system as a starting point. It then *mutates* the file system according to a *mutation profile*, which internally contains a multi-dimensional histogram. To create profiles, we analyzed data and metadata changes in several public and private datasets: home directories, system logs, email and Web servers, and a version control repository. The total size of our datasets approaches 10TB; the sum of observation periods exceeds one year, with the longest single dataset exceeding 6 months' worth of recordings.

Our framework is versatile, modular, and efficient. We use an in-memory file system tree that can be populated and mutated using a series of composable modules. Researchers can easily customize modules to emulate file system changes that the researchers observe. After all appropriate mutations are done, the in-memory tree can be quickly written to disk. For example, we generated a 4TB file system on a machine with a single drive in only 13 hours, 12 of which were spent writing data to the drive.

4.2 Previous Datasets

To quantify the lack of readily available and representative datasets, we surveyed 33 deduplication papers published in major conferences in 2000–2011: ten papers were in Usenix ATC, ten in Usenix FAST, four in SYSTOR, two in IEEE MSST, and the remaining seven elsewhere. We classified 120 datasets used in these papers as: (1) Private datasets accessible only to particular authors; (2) Public datasets which are hard or impossible to reproduce (e.g., CNN web-site snapshots on certain dates); (3) Artificially synthesized datasets; and (4) Public datasets that are easily reproducible by anyone.

We found that 29 papers (89%) used *at least one* private dataset for evaluation. The remaining four papers (11%) used artificially synthesized datasets, but details of the synthesis were omitted. This makes it nearly impossible to fairly compare many papers among the 33 surveyed (assuming that the authors do not have access to the same private datasets). Across all datasets, 64 (53%) were private, 17 (14%) were public but hard to reproduce, and 11 (9%) were synthetic datasets without generation details. In total, 76% of the datasets were unusable for cross-system evaluation. Of the 28 datasets (24%) we characterized as public, twenty were smaller than 1GB in logical size, much too small to evaluate any real deduplication system. The remaining eight datasets contained various operating system distributions in different formats: installed, ISO, or VM images.

Clearly, the few publicly available datasets do not adequately represent the entirety of real-world information. But releasing large real datasets is challenging for privacy reasons, and the sheer size of such datasets makes them unwieldy to distribute. Some researchers have suggested releasing hashes of files or file data rather than the data itself, to reduce the overall size of the released information and to avoid leaking private information. Unfortunately, hashes alone are insufficient: much effort goes into chunking algorithms, and there is no clear winning deduplication strategy because it often depends on the input data and workload being deduplicated.

4.3 Emulation Framework

In this section we first explain the generic approach we took for dataset generation and justify why it reflects many real-world situations. We then present the main components of our framework and their interactions. For the rest of the chapter, we use the term *metadata* to refer to the file system name-space (file names, types, sizes, directory depths, etc.), while *content* refers to the actual data within the files.

4.3.1 Generation Methods

Real-life file systems evolve over time as users and applications create, delete, copy, modify, and back up files. This activity produces several kinds of correlated information. Examples include 1) Identical downloaded files; 2) Users making copies by hand; 3) Source-control systems making copies; 4) Copies edited and modified by users and applications; 5) Full and partial backups repeatedly preserving the same files; and 6) Applications creating standard headers, footers, and templates.

To emulate real-world activity, one must account for all these sources of duplication. One option would be to carefully construct a statistical model that generates duplicate content. But it is difficult to build a statistics-driven system that can produce correlated output of the type needed for this project. We considered directly generating a file system containing duplicate content, but rejected the approach as impractical and non-scalable.

Instead, we emulate the evolution of real file systems. We begin with a simulated *initial snapshot* of the file system at a given time. (We use the term "snapshot" to refer to the complete state of a file system; our usage is distinct from the copy-on-write snapshotting technology available in some systems.) The initial snapshot can be based on a live file system or can be artificially generated by a system such as Impressions [2]. In either case, we *evolve* the snapshot over time by applying *mutations* that simulate the activities that generate both unique and duplicate content. Because our evolution is based on the way real users and applications change file systems, our approach is able to generate file systems and backup streams that accurately simulate real-world conditions, while offering the researcher the flexibility to tune various parameters to match a given situation.

Our mutation process can operate on file systems in two dimensions: space and time. The "space" dimension is equivalent to a single snapshot, and is useful to emulate deduplication in primary storage (e.g., if two users each have an identical copy of the same file). "Time" is equivalent to backup workloads, which are very common in deduplication systems, because snapshots are taken within some pre-defined interval (e.g., one day). Virtualized environments exhibit both dimensions, since users often create multiple virtual machines (VMs) with identical file systems that diverge over time because the VMs are used for different purposes. Our system lets researchers create mutators for representative VM user classes and generate appropriately evolved file systems.

Our system's ability to support logical changes in both space and time lets it evaluate deduplication for all major use cases: primary storage, backup, and virtualized environments.

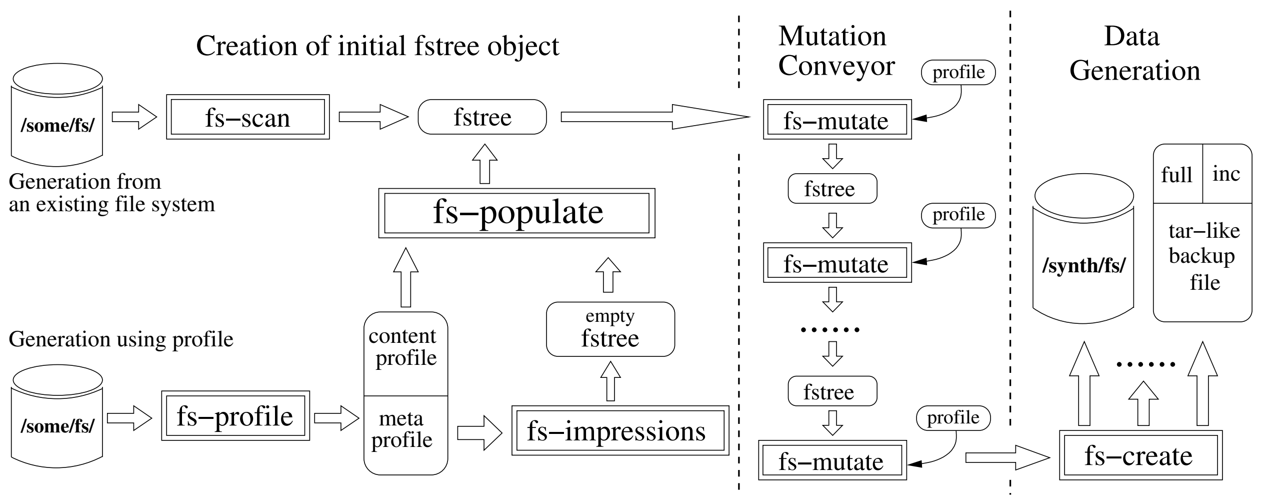


Figure 4.1: Action modules and their relationships. Double-boxed rectangles represent action modules and rectangles with rounded corners designate fstrees and other inputs and outputs. Left side of the figure describes the ways to create initial fstree object. Central part presents the mutation conveyor. Finally, the right side depicts several methods for writing out the final dataset.

4.3.2 Fstree Objects

Deduplication is usually applied to large datasets with hundreds of GB per snapshot and dozens of snapshots. Generating and repeatedly mutating a large file system would be unacceptably slow, so our framework performs most of its work without I/O. Output happens only at the end of the cycle when the actual file system is created.

To avoid excess I/O, we use a small in-memory representation—an *fstree*—that stores only the information needed for file system generation. This idea is borrowed from the design of Filebench [39]. The fstree contains pointers to *directory* and *file* objects. Each directory tracks its parent and a list of its files and sub-directories. The file object does not store the file's complete content; instead, we keep a list of its logical *chunks*, each of which has an identifier that corresponds to (but is not identical to) its deduplication hash. We later use the identifier to generate unique content for the chunk. If two chunks have same id means that the data in this chunk is the same. We use only 4 bytes for a chunk identifier, allowing up to 2^{32} unique chunks. Assuming a 50% deduplication ratio and a 4KB average chunk size, this can represent 32TB of storage. Note that a single fstree normally represents a single snapshot, so 32TB is enough for most modern datasets. For larger datasets, the identifier field can easily be expanded.

To save memory, we do not track per-object user or group IDs, permissions, or other properties. If this information is needed in a certain model (e.g., if some users modify their files more often than others), all objects have a variable-sized private section that can store any information required by a particular emulation model.

The total size of the fstree depends on the number of files, directories, and logical chunks. File, directory, and chunk objects are 29, 36, and 20 bytes, respectively. Representing a 2TB file system in which the average file was 16KB and the average directory held ten files would require 9GB of RAM. A server with 64GB could thus generate realistic 14TB file systems. Note that this is the size of a *single* snapshot, and in many deduplication studies one wants to look at 2–3 months worth of daily backups. In this case, one would write a snapshot after each fstree mutation and then continue with the same in-memory fstree. In such a scenario, our system is capable of producing datasets of much larger sizes; e.g., for 90 full backups we could generate 1.2PB of test data.

Our experience has shown that it is often useful to save fstree objects (the object, not the full file system) to persistent storage. This allows us to reuse an fstree in different ways, e.g., representing the behavior of different users in a multi-tenant cloud environment. We designed the fstree so that it can be efficiently serialized to or from disk using only a single sequential I/O. Thus it takes less than two minutes to save or load a 9GB fstree on a modern 100MB/sec disk drive. Using a disk array can make this even faster.

4.3.3 Fstree Action Modules

An fstree represents a static image of a file system tree—a snapshot. Our framework defines several operations on fstrees, which are implemented by pluggable *action modules*; Figure 4.1 demonstrates their relationships. Double-boxed rectangles represent action modules; rounded ones designate inputs and outputs.

FS-SCAN. One way to obtain an initial fstree object (to be synthetically modified later) is to scan an existing file system. The FS-SCAN module does this: it scans content and metadata, creates file, directory, and chunk objects, and populates per-file chunk lists. Different implementations of this module can collect different levels of detail about a file system, such as recognizing or ignoring symlinks, hardlinks, or sparse files, storing or skipping file permissions, using different chunking algorithms.

FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE. Often, an initial file system is not available, or cannot be released even in the form of an fstree due to sensitive data. FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE address this problem. FS-PROFILE is similar to FS-SCAN, but does not collect such detailed information, instead gathering only a statistical profile. The specific information collected depends on the implementation, but we assume it does not reveal sensitive data. We distinguish sub-parts: the *meta profile*, which contains statistics about the metadata, and the *content profile*.

Several existing tools can generate a static file system image based on a metadata profile [2, 39], and any of these can be reused by our system. A popular option is Impressions [2], which we modified to produce an fstree object instead of a file system image (FS-IMPRESSIONS). This fstree object is *empty*, meaning it contains no information about file contents. FS-POPULATE fills an empty fstree by creating chunks based on the content profile. Our current implementation takes the distribution of duplicates as a parameter; more sophisticated versions are not the focus of this thesis and are left for the future work. The left part of Figure 4.1 depicts the two current options for creating initial fstrees. This study focuses on the mutation module (below).

FS-MUTATE. FS-MUTATE is a key component of our approach. It mutates the fstree according to the changes observed in a real environment. Usually it iterates over all files and directories in the fstree and deletes, creates, or modifies them. A single mutation can represent weekly, daily, or hourly changes; updates produced by one or more users; etc. FS-MUTATE modules can be chained as shown in Figure 4.1 to represent multiple changes corresponding to different users, different

Name	Total size (GB)	Total files (thousands)	Snapshots & period	Avg. snapshot size (GB)	Avg. number of files in a snapshot (thousands)
Kernels	13	903	40	0.3	23
CentOS	36	1,559	8	4.5	195
Home	3,482	15,352	15 weekly	227	1,023
MacOS	4,080	83,220	71 daily	59	1,173
System Logs	626	2,672	8 weekly	78	334
Sources	1,331	1,112	8 weekly	162	139

Table 4.1: Summary of analyzed datasets.

times, etc. Usually, a mutation module is controlled by a parameterized profile based on real-world observations. The profile can also be chosen to allow micro-benchmarking, such as varying the percentage of unique chunks to observe changes in deduplication behavior. In addition, if a profile characterizes the changes between an empty file system and a populated one, FS-MUTATE can be used to generate an initial file system snapshot.

FS-CREATE. After all mutations are performed, FS-CREATE generates a final dataset in the form needed by a particular deduplication system. In the most common case, FS-CREATE produces a file system by walking through all objects, creating the corresponding directories and files, and generating file contents based on the chunk identifiers. Content generation is implementation-specific; for example, contents might depend on the file type or on an entropy level. The important property to preserve is that the same chunk identifiers result in the same content, and different chunk identifiers produce different content. FS-CREATE could also generate tar-like files for input to a backup system, which can be significantly faster than creating a complete file system because it can use sequential writes. FS-CREATE could also generate only the files that have changed since the previous snapshot, emulating data coming from an incremental backup.

4.3.4 Usage Example

To benchmark a deduplication system using our framework, a user needs to complete the following steps. First, the initial fstree object is either obtained from a dataset distributor or by scanning a local file system. Dataset distributor is an entity that owns the original datasets and distributes them in the form of fstree objects and mutation modules. Second, the user obtains FS-MUTATE module implementation (in the form of executable) from the distributor. Third, the user runs FS-MUTATE module against the initial and subsequent fstree objects (mutation conveyor on Figure 4.1). The length of conveyor and its structure depends on the specifics of the dataset against which the deduplication system is evaluated (e.g., depending on the backup policy). Fourth, the actual data is generated in a desired format using FS-CREATE module. Finally, resulting dataset is fed to the deduplication system to measure its performance.

4.4 Datasets Analyzed

To create a specific implementation of the framework modules, we analyzed file system changes in six different datasets; in each case, we used FS-SCAN to collect hashes and file system tree characteristics. We chose two commonly used public datasets, two collected locally, and two originally presented by Dong et al. [33].

Table 4.1 describes important characteristics of our six datasets: total size, number of files, and per-snapshot averages. Our largest dataset, MacOS, is 4TB in size and has 83 million files spanning 71 days of snapshots.

Kernels: Unpacked Linux kernel sources from version 2.6.0 to version 2.6.39.

CentOS: Complete installations of eight different releases of the CentOS Linux distribution from version 5.0 to 5.7.

Home: Weekly snapshots of students' home directories from a shared file system. The files consisted of source code, binaries, office documents, virtual machine images, and miscellaneous files.

MacOS: A Mac OS X Enterprise Server that hosts various services for our research group: email, mailing lists, Web-servers, wiki, Bugzilla, CUPS server, and an RT trouble-ticketing server.

System Logs: Weekly unpacked backups of a server's `/var` directory, mostly consisting of emails stored by a list server.

Sources: Weekly unpacked backups of source code and change logs from a Perforce version control repository.

Of course, different environments can produce significantly different datasets. For that reason, our design is flexible, and our prototype modules are parameterized by profiles that describe the characteristics of a particular dataset's changes. If necessary, other researchers can use our profile collector to gather appropriate distributions, or implement a different FS-MUTATE model to express the changes observed in a specific environment.

4.5 Module Implementations

There are many ways to implement our framework's modules. Each corresponds to a model that describes a dataset's behavior in a certain environment. An ideal model should capture the characteristics that most affect the behavior of a deduplication system. In this section we first explore the space of parameters that can affect the performance of a deduplication system, and then present a model for emulating our datasets' behavior. Our implementation can be downloaded from `https://avatar.fsl.cs.sunysb.edu/groups/deduplicationpublic/`.

4.5.1 Space Characteristics

Both content and metadata characteristics are important for accurate evaluation of deduplication systems. Figure 4.2 shows a rough classification of relevant dataset characteristics. Previous research has primarily focused on characterizing *static* file system snapshots [2]. Instead, we are interested in characterizing the file system's *dynamic* properties (both content and metadata). Extending the analysis to multiple snapshots can give us information about file deletions, creations, and modifications. This in turn will reflect on the properties of static snapshots.

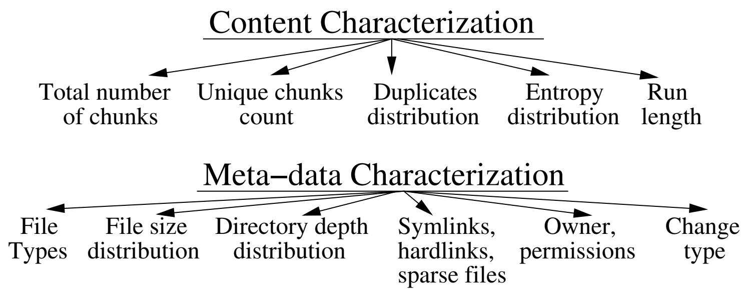


Figure 4.2: Content and metadata characteristics of file systems that are relevant to deduplication system performance.

Any deduplication solution divides a dataset into chunks of fixed or variable size, indexes their hashes, and compares new incoming chunks against the index. If a new hash is already present, the duplicate chunk is discarded and a mapping that allows the required data to be located later is updated.

Therefore, the total number of chunks and the number of unique chunks in a dataset affects the system's performance. The performance of some data structures used in deduplication systems also depends on the distribution of duplicates, including the percentage of chunks with a certain number of duplicates and even the ordering of duplicates. E.g., it is faster to keep the index of hashes in RAM, but for large datasets a RAM index may be economically infeasible. Thus, many deduplication systems use sophisticated index caches and Bloom filters [148] to reduce RAM costs, complicating performance analysis.

For many systems, it is also important to capture the entropy distribution inside the chunks, because most deduplication systems support local chunk compression to further reduce space. Compression can be enabled or disabled depending on the data type [73].

A deduplication system's performance depends not only on content, but also on the file system's *metadata*. When one measures the performance of a conventional file system (without deduplication), the file size distribution and directory depth strongly impact the results [3]. Deduplication is sometimes an addition to existing conventional storage, in which case file sizes and directory depth will also affect the overall system performance.

The run lengths of unique or duplicated chunks can also be relevant. If unique chunks follow each other closely (in space and time), the storage system's I/O queues can fill up and throughput can drop. Run lengths depend on the ways files are modified: pure extension, as in log files; simple insertion, as for some text files; or complete rewrites, as in many binary files. Run lengths can also be indirectly affected by file size distributions, because it often happens that only a few files in the dataset change from one backup to another, and the distance between changed chunks within a backup stream depends on the sizes of the unchanged files.

Content-aware deduplication systems sometimes use the file header to detect file types and improve chunking; others use file owners or permissions to adjust their deduplication algorithms.

Finally, symlinks, hardlinks, and sparse files are a rudimentary form of deduplication, and their presence in a dataset can affect deduplication ratios.

Dependencies. An additional issue is that many of the parameters mentioned above depend on each other, so considering their statistical distributions independently is not possible. For example, imagine that emulating the changes to a specific snapshot requires removing N files. We also want the total number of chunks to be realistic, so we need to remove files of an appropriate size. Moreover, the distribution of duplicates needs to be preserved, so the files that are removed should contain the appropriate number of unique and duplicated chunks. Preserving such dependencies is important, and our FS-MUTATE implementation (presented next) allows that.

4.5.2 Markov & Distribution (M&D) Model

We call our model *M&D* because it is based on two abstractions: a Markov model for classifying file changes, and a multi-dimensional histogram representing statistical dependencies between file characteristics.

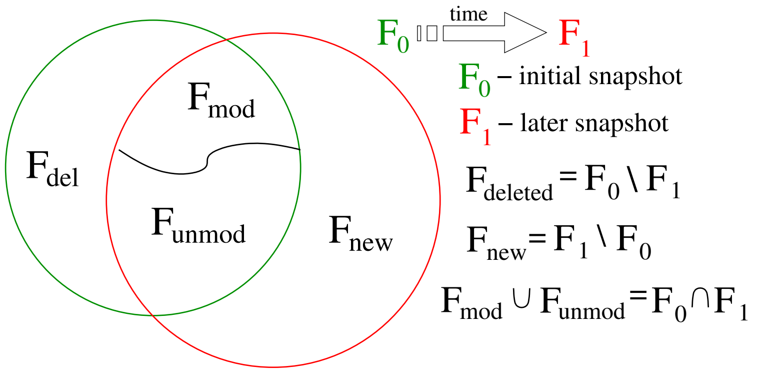


Figure 4.3: Classification of files. F_0 and F_1 are files from two subsequent snapshots.

Markov model. Suppose we have two snapshots of a file system taken at two points in time: F_0 and F_1. We classify files in F_0 and F_1 into four sets: 1) F_{del}: files that exist in F_0, but are missing in F_1. 2) F_{new}: files that exist in F_1, but are missing in F_0. 3) F_{mod}: files that exist in both F_0 and F_1, but were modified. 4) F_{unmod}: files in F_0 and F_1 that were not modified. The relationship between these sets is depicted in Figure 4.3. In our study, we identify files by their full pathname, i.e., a file in the second snapshot with the same pathname as one in the first is assumed to be a later version of the same file.

Analysis of our datasets showed that the file sets defined above remain relatively stable. Files that were unmodified between snapshots $F_0 \rightarrow F_1$ tended to remain unmodified between snapshots $F_1 \rightarrow F_2$. However, files still migrate between sets, with different rates for different datasets. To capture such behavior we use the Markov model depicted in Figure 4.4. Each file in the fstree has a state assigned to it in accordance with the classification defined earlier. In the fstree representing the first snapshot, all files have the New state. Then, during mutation, the file states change with precalculated probabilities that have been extracted by looking at a window of three real snapshots, covering two file transitions: between the first and second snapshots and between the second and third ones. This is the minimum required to allow us to calculate conditional probabilities for the Markov model. For example, if some file is modified between snapshots $F_0 \rightarrow F_1$ and is also modified in $F_1 \rightarrow F_2$, then this is a Modified→Modified (MM) transition. Counting the number of

Dataset	**N**	**NM**	**NU**	**ND**	**MU**	**MD**	**MM**	**UM**	**UD**	**UU**	**DN**	**D**
Kernels	5	32	65	3	49	3	48	17	3	80	1	3
CentOS	13	4	22	74	43	2	55	4	1	95	1	10
Home	4	2	78	20	54	10	36	0.14	0.35	99.51	6	0.50
MacOS	0.1	11	78	11	37.46	0.03	62.51	0.05	0.03	99.92	1	0.03
System Logs	2	9	90	1	44.40	0.18	55.42	0.03	0.01	99.06	4	0.02
Sources	0.2	7	88	5	58.76	0.04	41.20	0.07	0	99.93	0	0.01

Table 4.2: Probabilities (in percents) of file state transitions for different datasets. N: new file appearance. D: file deletion.
NM: New→Modified transition. NU: New→Unmodified transition. ND: New→Deleted transition, etc.

MM transitions among the total number of state transitions allows us to compute the corresponding probability; we did this for each possible transition.

Some transitions, such as Deleted→New (DN), may seem counterintuitive. However, some files are recreated after being deleted, producing nonzero probabilities for this transition. Similarly, if a file is renamed or moved, it will be counted as two transitions: a removal and a creation. In this case, we allocate duplicated chunks to the new file in a later stage.

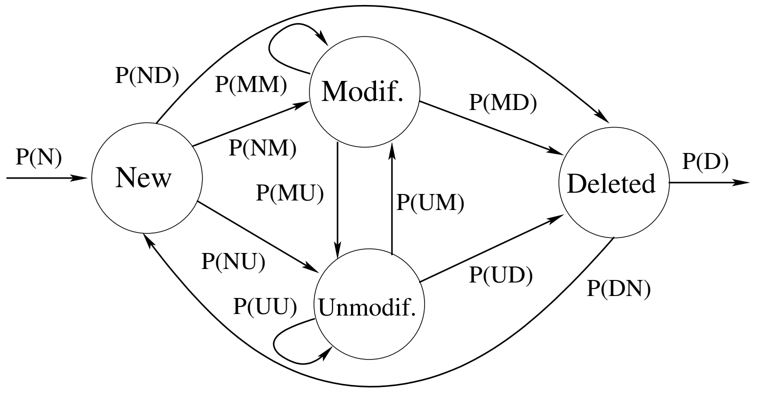


Figure 4.4: Markov model for handling file states. State transitions are denoted by the first letters of the source and destination states. For example, NM denotes a New→Modified transition and P(NM) is the transition's probability.

The Markov model allows us to accurately capture the rates of file appearance, deletion, and modification in the trace. Table 4.2 presents the average transition probabilities observed for our datasets. As mentioned earlier, in all datasets files often remain Unchanged, and thus the probabilities of UU transitions are high. The chances for a changed file to be re-modified are around 50% for many of our datasets. The probabilities for many other transitions vary significantly across different datasets.

Multi-dimensional histogram. When we analyzed real snapshots, we collected three multi-dimensional file histograms: $M_{del}(p_1, ..., p_{n_{del}})$, $M_{new}(p_1, ..., p_{n_{new}})$, and $M_{mod}(p_1, ..., p_{n_{mod}})$ for deleted, new, and modified files, respectively. The parameters of these histograms (p_1, ..., p_n) represent the characteristics of the files that were deleted, created, or modified. As described in Section 4.5.1, many factors affect deduplication. In this work, we selected several that we deemed most relevant for a generic deduplication system. However, the organization of our FS-MUTATE module allows the list of emulated characteristics to be easily extended.

All three histograms include these parameters:

depth: directory depth of a file;

ext: file extension;

size: file size (in chunks):

uniq: the number of chunks in a file that are not present in the previous snapshot (i.e., unique chunks);

dup1: the number of chunks in a file that have only one duplicate in the entire previous snapshot; and

dup2: the number of chunks in a file that occur exactly twice in the entire previous snapshot.

We consider only the chunks that occur up to 3 times in a snapshot because in all our snapshots these chunks constituted more than 96% of all chunks.

During mutation, we use the histogram of new files:

$$M_{new}(\mathit{depth}, \mathit{ext}, \mathit{size}, \mathit{uniq}, \mathit{dup1}, \mathit{dup2})$$

to create the required number of files with the appropriate properties. E.g., if $M_{new}(2, \text{“}.c\text{”}, 7, 3, 1, 1)$ equals four, then FS-MUTATE creates four files with a "`.c`" extension at directory depth two. The size of the created files is seven chunks, of which three are unique, one has a single duplicate, and one has two duplicates across the entire snapshot. The hashes for the remaining two chunks are selected using a per-snapshot (not per-file) distribution of duplicates, which is collected during analysis along with M_{new}. Recall that FS-MUTATE does not generate the content of the chunks, but only their hashes. Later, during on-disk snapshot creation, FS-CREATE will generate the content based on the hashes.

When selecting files for deletion, FS-MUTATE uses the deleted-files histogram:

$$M_{del}(\mathit{depth}, \mathit{ext}, \mathit{size}, \mathit{uniq}, \mathit{dup1}, \mathit{dup2}, \mathit{state})$$

This contains an additional parameter—*state*—that allows us to elegantly incorporate a Markov model in the histogram. The value of this parameter can be one of the Markov states New, Modified, Unmodified, or Deleted; we maintain the state of each file within the fstree. A file is created in the New state; later, if FS-MUTATE modifies it, its state is changed to Modified; otherwise it becomes Unmodified. When FS-MUTATE selects files for deletion, it limits its search to files in the state given by the corresponding M_{del} entry. For example, if $M_{del}(2, \text{“}.c\text{”}, 7, 3, 1, 1, \text{“}\mathit{Modified}\text{”})$ equals one, then FS-MUTATE tries to delete a single file in the Modified state (all other parameters should match as well).

To select files for modification, FS-MUTATE uses the M_{mod} distribution, which has the same parameters as M_{del}. But unlike deleted files, FS-MUTATE needs to decide *how* to change the files. For every entry in M_{mod}, we keep a list of *change descriptors*, each of which contains the file's characteristics *after* modification:

1. File size (in chunks);

2. The number of unique chunks (here and in the two items below, duplicates are counted against the entire snapshot);

Dataset	B	E	M	BE	BM	ME	BEM
Kernels	52	8	7	14	5	3	11
CentOS	69	3	2	8	2	1	15
Home	38	3	8	10	11	1	29
MacOS	53	21	1	12	1	1	11
Sys. Logs	42	34	5	6	0	1	10
Sources	20	6	41	7	7	1	18

Table 4.3: Probabilities of the change patterns for different datasets (in percents).

3. The number of chunks with one duplicate;
4. The number of chunks with two duplicates; and
5. Change pattern.

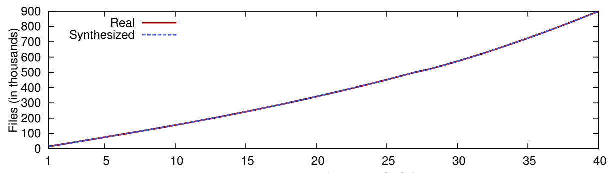
All parameters except the last are self-explanatory. The change pattern encodes the way a file was modified. We currently support the following three options: B—the file was modified in the beginning (this usually corresponds to prepend); E—the file was modified at the end (corresponds to file extension or truncation); and M—the file was modified somewhere in the middle, which corresponds to the case when neither the first nor the last chunk were modified, but others have changed. We also support combinations of these patterns: *BE*, *BM*, *EM*, and *BEM*. To recognize the change pattern during analysis, we sample the corresponding chunks in the old and new files. Table 4.3 presents the average change patterns for different datasets. For all datasets the number of files modified in the beginning is high. This is a consequence of chunk-based analysis: files that are smaller than the chunk size contain a single chunk. Therefore, wherever small files are modified, the first (and only) chunk differs in two subsequent versions, which our analysis identifies as a change in the file's beginning. For the System Logs dataset, the number of files modified at the end is high because logs are usually appended. In the Sources dataset many files are modified in the middle, which corresponds to small patches in the code.

We collect change descriptors and the M_{mod} distribution during the analysis phase. During mutation, when a file is selected for modification using M_{mod}, one of the aforementioned change descriptors is selected randomly and the appropriate changes are applied.

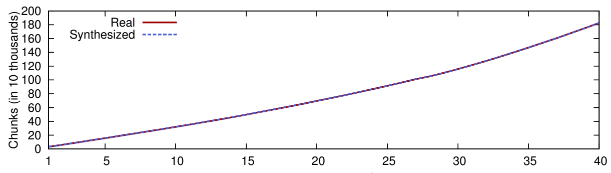
It is possible that the number of files that satisfy the distribution parameters is larger than the number that need to be deleted or modified. In this case, FS-MUTATE randomly selects files to operate on. If not enough files with the required properties are in the fstree, then FS-MUTATE tries to find the best match based on a simple heuristic: the file that matches most of the properties. Other definitions of best match are possible, and we plan to experiment with this parameter in the future.

Multi-dimensional histograms capture not only the statistical frequency of various parameters, but also their interdependencies. By adding more distribution dimensions, one can easily emulate other parameters.

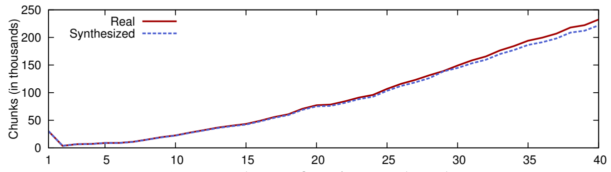
Analysis. To create profiles for our datasets, we first scanned them using the FS-SCAN module mentioned previously. We use variable chunking with an 8KB average size; variable chunking is needed to properly detect the type of file change, since prepended data causes fixed-chunking



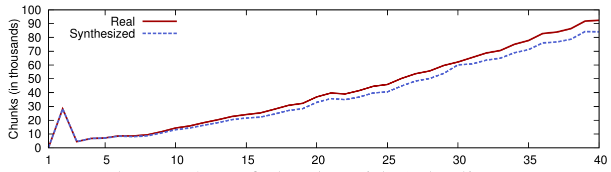
(a) Total number of files



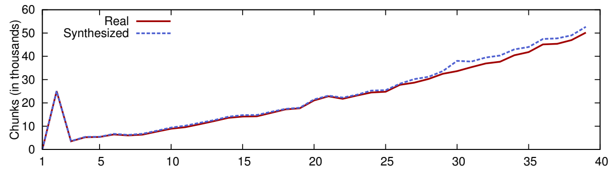
(b) Total number of chunks



(c) Number of unique chunks



(d) Number of chunks with 1 duplicate



(e) Number of chunks with 2 duplicates

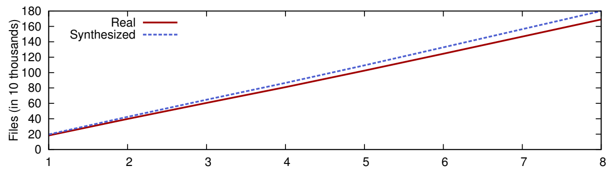
Figure 4.5: Emulated parameters for Kernels real and synthesized datasets as the number of snapshots in them increases.

systems to see a change in every chunk. We chose 8KB as a compromise between accuracy (smaller sizes are more accurate) and the speed of the analysis, mutation, and file system creation steps.

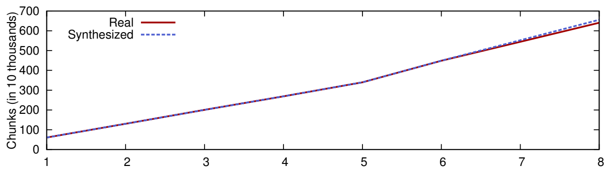
The information collected by FS-SCAN was loaded into a database; we then used SQL queries to extract distributions. The analysis of our smallest dataset (Kernels) took less than 2 hours, whereas the largest dataset (MacOS) took about 45 hours of wall-clock time on a single workstation. This analysis can be sped up by parallelizing it. However, since it needs to be done only once to extract a profile, a moderately lengthy computation is acceptable. Mutation and generation of a file system run much faster and are evaluated in Section 4.6. The size of the resulting profiles varied from 8KB to 300KB depending on the number of changes in the dataset.

Chunk generation. Our FS-CREATE implementation generates chunk content by maintaining a randomly generated buffer. Before writing a chunk to the disk, this buffer is XORed with the chunk ID to ensure that each ID produces a unique chunk and that duplicates have the same content. We currently do not preserve the chunk's entropy because our scan tool does not yet collect this data. FS-SCAN collects the size of every chunk, which is kept in the in-memory fstree object for use by FS-CREATE. New chunks in mutated snapshots have their size set by FS-MUTATE according to a per-snapshot chunk-size distribution. However, deduplication systems can use *any* chunk size that is larger than or equal to the one that FS-SCAN uses. In fact, sequences of identical chunks may appear in several subsequent snapshots. As these sequences of chunks are relatively long, any chunking algorithm can detect an appropriate number of identical chunks across several snapshots.

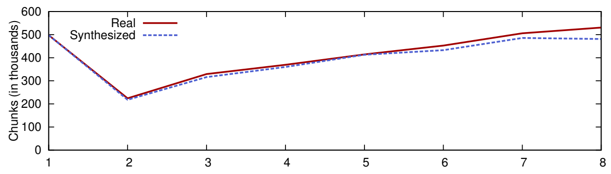
Security guarantees. The FS-SCAN tool uses 48-bit fingerprints, which are prefixes of 16 byte MD5 hashes; this provides a good level of security, although we may be open to dictionary attacks. Stronger anonymization forms can be easily added in the future work.



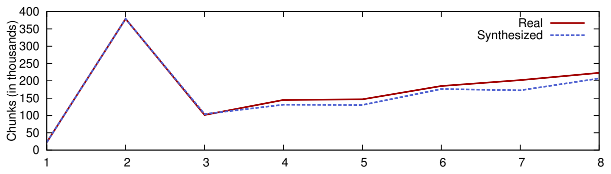
(a) Total number of files



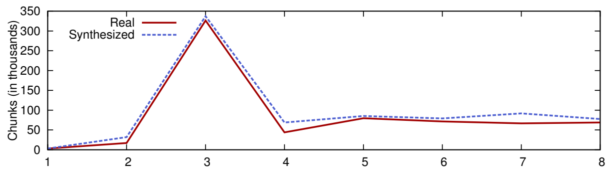
(b) Total number of chunks



(c) Number of unique chunks

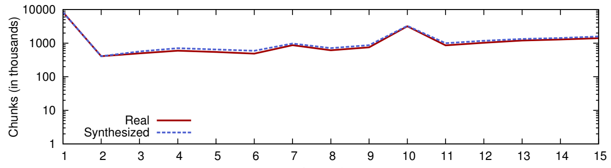


(d) Number of chunks with 1 duplicate

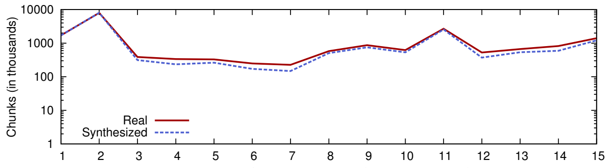


(e) Number of chunks with 2 duplicates

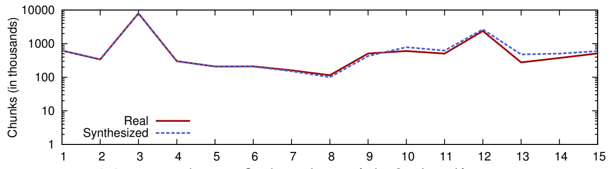
Figure 4.6: Emulated parameters for CentOS real and synthesized datasets as the number of snapshots in them increases.



(a) Number of unique chunks



(b) Number of chunks with 1 duplicate



(c) Number of chunks with 2 duplicates

Figure 4.7: Emulated parameters for Homes real and synthesized datasets as the number of snapshots in them increases.

4.6 Evaluation

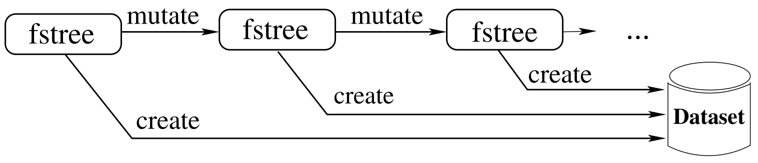
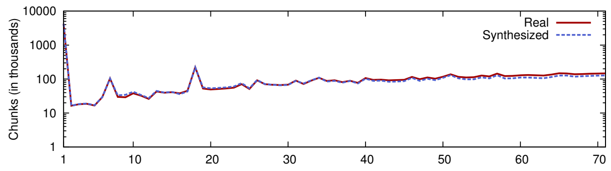
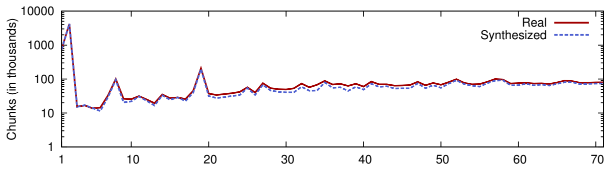


Figure 4.8: The process of dataset formation.

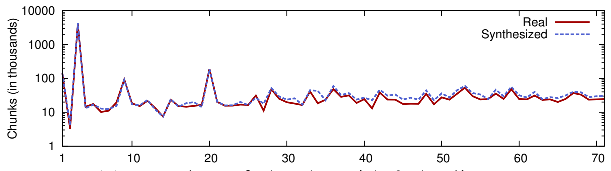
We collected profiles for the datasets described in Section 4.4 and generated the same number of synthetic snapshots as the real datasets had, chaining different invocations of FS-MUTATE so that the output of one mutation served as input to the next. All synthesized snapshots together form a synthetic dataset that corresponds to the whole real dataset (Figure 4.8). We generated



(a) Number of unique chunks

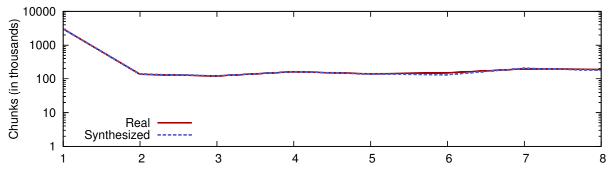


(b) Number of chunks with 1 duplicate

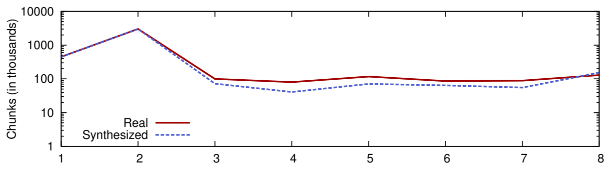


(c) Number of chunks with 2 duplicates

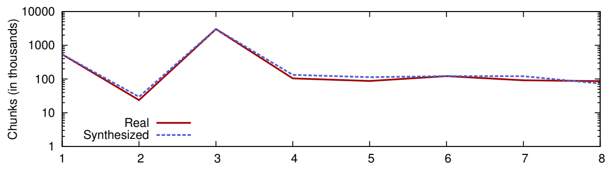
Figure 4.9: Emulated parameters for MacOS real and synthesized datasets as the number of snapshots in them increases.



(a) Number of unique chunks



(b) Number of chunks with 1 duplicate



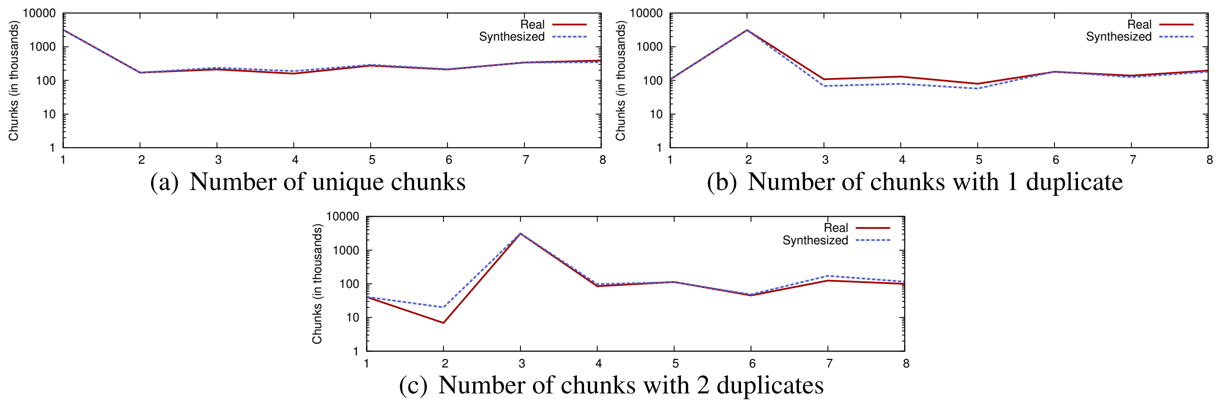
(c) Number of chunks with 2 duplicates

Figure 4.10: Emulated parameters for System Logs real and synthesized datasets as the number of snapshots in them increases.

the initial fstree object by running FS-SCAN on the real file system. Each time a new snapshot was added, we measured the total files, total chunks, numbers of unique chunks and those that had one and two duplicates, directory depth, file size and file type distributions.

First, we evaluated the parameters that FS-MUTATE emulates. Figures 4.5–4.11 contain the graphs for the real and synthesized Kernels, CentOS, Homes, MacOS, System Logs, and Sources datasets, in order. The Y axis scale is linear for the Kernels and Sources datasets (Figures 4.5–4.6) and logarithmic for the others (Figures 4.7–4.11). We present file and chunk count graphs only for the Kernels and CentOS datasets. The relative error of these two parameters is less than 1% for all datasets, and the graphs look very similar: monotonic close-to-linear growth. The file count is insensitive to modification operations because files are not created or removed, which explains its high accuracy. The total chunk count is maintained because we carefully preserve file size during creation, modification, and deletion.

For all datasets the trends observed in the real data are closely followed by the synthesized data. However, certain discrepancies exist. Some of the steps in our FS-MUTATE module are random; e.g., the files deleted or modified are not precisely the same ones as in the real snapshot, but instead ones with similar properties. This means that our synthetic snapshots might not have the same files



(a) Number of unique chunks (b) Number of chunks with 1 duplicate (c) Number of chunks with 2 duplicates

Figure 4.11: Emulated parameters for Sources real and synthesized datasets as the number of snapshots in them increases.

Dataset	Files	Chunks	Unique chunks	1 Dup. chunks	2 Dup. chunks
Kernels	< 1	< 1	4	9	5
CentOS	6	2	9	7	11
Home	< 1	< 1	12	13	14
MacOS	< 1	< 1	4	9	4
Sys. Logs	< 1	< 1	6	15	15
Sources	< 1	< 1	10	8	13

Table 4.4: Relative error of emulated parameters after the final run for different datasets (in percents).

that would exist in the real snapshot. As a result, FS-MUTATE cannot find some files during the following mutations and so the best-match strategy is used, contributing to the *instantaneous* error of our method. However, because our random actions are controlled by the real statistics, the deviation is limited in the long run.

The graphs for unique chunks have an initial peak because there is only one snapshot at first, and there are not many duplicates in a single snapshot. As expected, this peak moves to the right in the graphs for chunks with one and two duplicates.

The Homes dataset has a second peak in all graphs around 10–12 snapshots (Figure 4.7). This point corresponds to two missing weekly snapshots. The first was missed due to a power outage; the second was missed because our scan did not recover properly from the power outage. As a result, the 10th snapshot contributes many more unique chunks in the dataset than the others.

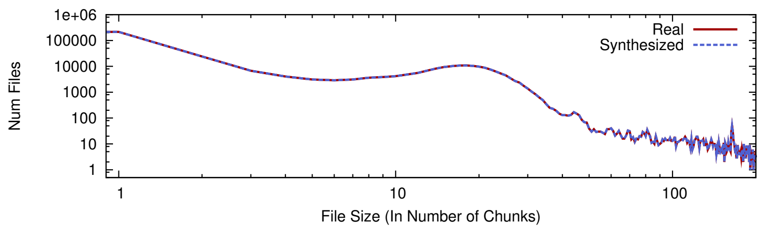
The MacOS dataset contains daily, not weekly snapshots. Daily changes in the system are more sporadic than weekly ones: one day users and applications add a lot of new data, the next many files are copied, etc. Figure 4.9 therefore contains many small variations.

Table 4.4 shows the relative error for emulated parameters at the end of each run. Maximum deviation did not exceed 15% and averaged 6% for all parameters and datasets. We also analyzed the file size, type, and directory depth distributions in the final dataset. Figure 4.12 demonstrates these for several representative datasets. In all cases the error was fairly low, within 2%.

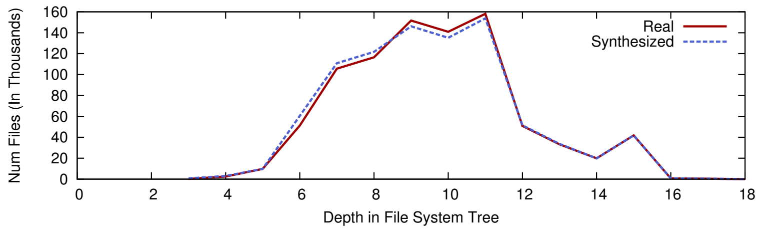
Dataset	Total size (GB)	Snap-shots	Mutat. time	Creat. time	Total time
Kernels	13	40	30 sec	6 sec	5 min
CentOS	36	8	3 min	95 sec	13 min
Home	3,482	15	44 min	38 min	10 hr
MacOS	4,080	71	49 min	10 min	13 hr
Sys. Logs	626	8	14 min	4 hr	32 hr
Sources	1,331	8	21 min	4 hr	32 hr

Table 4.5: Times to mutate and generate data sets.

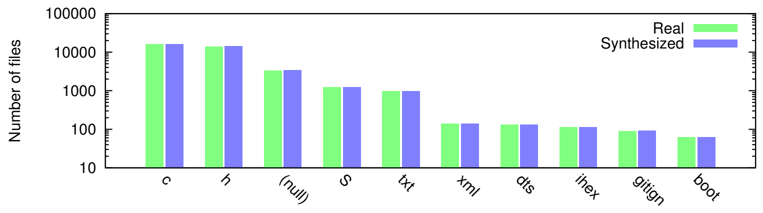
The snapshots in our datasets change a lot. For example, the deduplication ratio is less than 5 in our Kernels dataset, even though the number of snapshots is 40. We expect the accuracy of our system to be higher for the datasets that change slower; for instance, datasets with identical snapshots are emulated without any error.



(a) File size distribution (System Logs)



(b) Directory depth distribution (Homes)



(c) File type distribution (Kernels)

Figure 4.12: File size, type, and directory depth distributions for different real and synthesized dataset.

Performance. We measured the time of every mutation and creation operation in the experiments above. The Kernels, CentOS, Home, and MacOS experiments were conducted on a machine with an Intel Xeon X5680 3.3GHz CPU and 64GB of RAM. The snapshots were written to a single Seagate Savvio 15K RPM disk drive. For some datasets the disk drive could not hold all the snapshots, so we removed them after running FS-SCAN for accuracy analysis. Due to privacy constraints the System Logs and Sources experiments were run on a different machine with an AMD Opteron 2216 2.4GHz CPU, 32GB of RAM, and a Seagate Barracuda 7,200 RPM disk drive. Unfortunately, we had to share the second machine with a long-running job that periodically performed random disk reads.

Table 4.5 shows the *total* mutation time for all snapshots, the time to write a *single* snapshot to the disk, and the total time to perform all mutations plus write the whole dataset to the disk. The creation time includes the time to write to disk. For convenience the table also contains dataset sizes and snapshot counts.

Even for the largest dataset, we completed all mutations within one hour; dataset size is the major factor in mutation time. Creation time is mostly limited by the underlying system's performance: the creation throughput of the Home and MacOS datasets is almost twice that of Kernels and CentOS, because the average file size is 2–10$\times$ larger for the former datasets, exploiting the high sequential drive throughput. The creation time was significantly increased on the second system because of a slower disk drive (7,200RPM vs. 15KRPM) and the interfering job, contributing to the 32-hour run time.

For the datasets that can fit in RAM—CentOS and Kernels—we performed an additional FS-CREATE run so that it creates data on tmpfs. The throughput in both cases approached 1GB/sec, indicating that our chunk generation algorithm does not incur much overhead.

4.7 Related Work

A number of studies have characterized file system workloads using I/O traces [77, 82, 101, 106] that contain information about all I/O requests observed during a certain period. The duration of a full trace is usually limited to several days, which makes it hard to analyze long-term file system changes. Trace-based studies typically focus on the dynamic properties of the workload, such as I/O size, read-to-write ratio, etc., rather than file content as is needed for deduplication studies.

Many papers have used snapshots to characterize various file system properties [3, 15, 108, 132]. With the exception of Agrawal et al.'s study [3], discussed below, the papers examine only a single snapshot, so only static properties can be extracted and analyzed. Because conventional file systems are sensitive to metadata characteristics, snapshot-based studies focus on size distributions, directory depths or widths, and file types (derived from extensions). File and block lifetimes are analyzed based on timestamps [3, 15, 132]. Authors often discuss the correlation between file properties, e.g., size and type [15, 108]. Several studies have proposed high-level explanations for file size distributions and designed models for synthesizing specific distributions [35, 108]. Downey and Mitzenmacher explain a file size distribution in World Wide Web and present models for generating appropriate files [35, 94] Vogels and Douceur et al. performed the analysis for Windows snapshots [34, 132].

Less attention has been given to the analysis of long-term file system changes. Agrawal et al. examined the trends in file system characteristics from 2000–2004 [3]. The authors presented only metadata evolution: file count, size, type, age, and directory width and depth.

Some researchers have worked on artificial file system aging [2, 112] to emulate the fragmentation encountered in real long-lived file systems. Our mutation module modifies the file system in RAM and thus does not emulate file system fragmentation. Modeling fragmentation can be added in the future if it proves to impact deduplication systems' performance significantly.

A number of newer studies characterized deduplication ratios for various datasets. Meyer and Bolosky studied content and metadata in primary storage [92]. The authors collected file system content from over 800 computers and analyzed the deduplication ratios of different algorithms: whole-file, fixed chunking, and variable chunking. Several researchers characterized deduplication in backup storage [90, 103, 133] and for virtual machine disk images [65, 85]. Chamness presented a model for storage-capacity planning that accounts for the number of duplicates in backups [26]. None of these projects attempted to synthesize datasets with realistic properties.

File system benchmarks usually create a test file system from scratch. For example, in Filebench [39] one can specify file size and directory depth distributions for the creation phase, but the data written is either all zeros or random. Agrawal et al. presented a more detailed attempt to approximate the distributions encountered in real-world file systems [2]. Again, no attention was given in their study to generating duplicated content.

4.8 Conclusions

Researchers and companies evaluate deduplication with a variety of datasets that in most cases are private, unrepresentative, or small in size. As a result, the community lacks the resources needed for fair and versatile comparison. Our work has two key contributions.

First, we designed and implemented a generic framework that can emulate the formation of datasets in different scenarios. By implementing new mutation modules, organizations can expose the behavior of their internal datasets without releasing the actual data. Other groups can then regenerate comparable data and evaluate different deduplication solutions. Our framework is also suitable for controllable micro-benchmarking of deduplication solutions. It can generate arbitrarily large datasets while still preserving the original's relevant properties.

Second, we presented a specific implementation of the mutation module that emulates the behavior of several real-world datasets. To capture the metadata and content characteristics of the datasets, we used a hybrid Markov and Multi-Dimensional Histogram (MDH) model that has a low error rate—less than 15% during 8 to 71 mutations for all datasets. We release the datasets, tools and some of profiles described in this study so that organizations can perform comparable studies of deduplication systems. They can be downloaded from `https://avatar.fsl.cs.sunysb.edu/groups/deduplicationpublic/`. These powerful tools will help both industry and research to make intelligent decisions when selecting the right deduplication solutions for their specific environments.

Chapter 5

NAS Workloads in Virtualized Setups

5.1 Introduction

By the end of 2012 almost half of all applications running on x86 servers will be virtualized; in 2014 this number is projected to be close to 70% [16, 17]. Virtualization, if applied properly, can significantly improve system utilization, reduce management costs, and increase system reliability and scalability. With all the benefits of virtualization, managing the growth and scalability of storage is emerging as a major challenge.

In recent years, growth in network-based storage has outpaced that of direct-attached disks; by 2014 more than 90% of enterprise storage capacity is expected to be served by Network Attached Storage (NAS) and Storage Area Networks (SAN) [143]. Network-based storage can improve availability and scalability by providing shared access to large amounts of data. Within the network-based storage market, NAS capacity is predicted to increase at an annual growth rate of 60%, as compared to only 22% for SAN [128]. This faster NAS growth is explained in part by its lower cost and its convenient file system interface, which is richer, easier to manage, and more flexible than the block-level SAN interface.

The rapid expansion of virtualization and NAS has lead to explosive growth in the number of virtual disk images being stored on NAS servers. Encapsulating file systems in virtual disk image files simplifies the implementation of VM migration, cloning, and snapshotting—the features that naturally map to existing NAS functions. In addition, non-virtualized hosts can co-exist with virtualized hosts that use the same NAS interface, which permits a gradual migration of services from physical to virtual machines.

Storage performance plays a crucial role when administrators select the best NAS for their environment. One traditional way to evaluate NAS performance is to run a file system benchmark, such as SPECsfs2008 [116]. Vendors periodically submit the results of SPECsfs2008 to SPEC; the most recent submission was in November 2012. Because widely publicized benchmarks such as SPECsfs2008 figure so prominently in configuration and purchase decisions, it is essential to ensure that the workloads the benchmarks generate represent what is observed in real data centers.

This chapter makes two contributions: an analysis of changing virtualized NAS workloads, and the design and implementation of a system to generate realistic virtualized NAS workloads. We first demonstrate that the workloads generated by many current file system benchmarks do not represent the actual workloads produced by VMs. This in turn leads to a situation where

NFS procedures	Physical clients		Virtualized clients
	SPECsfs2008	Filebench	
Data	28%	36%	99%
Metadata	72%	64%	<1%

Table 5.1: The striking differences between virtualized and physical workloads for two benchmarks: SPECsfs2008 and Filebench (Web-server profile). Data operations include READ and WRITE. All other operations (e.g., CREATE, GETATTR, READDIR) are characterized as metadata.

the performance results of a benchmark deviate significantly from the performance observed in real-world deployments. Although benchmarks are never perfect models of real workloads, the introduction of VMs has exacerbated the problem significantly. Consider just one example, the percentage of data and metadata operations generated by physical and virtualized clients. Table 5.1 presents the results for the SPECsfs2008 and Filebench web-server benchmarks that attempt to provide a "realistic" mix of metadata and data operations. We see that metadata procedures, which dominated in physical workloads, are almost non-existent when VMs are utilized. The reason is that, VMs store their guest file system inside large disk image files, mainly for convenience. Consequently, all metadata operations (and indeed all data operations) from the applications are converted into simple reads and writes to the image file.

Metadata-to-data conversion is just one example of the way workloads shift when virtual machines are introduced. In this study we examine, by collecting and analyzing a set of I/O traces generated by current benchmarks, how NAS workloads change when used in virtualized environments. We then leverage multi-dimensional trace analysis techniques from Chapter 3 to convert these traces to benchmarks.

Previously, most file system benchmarks used simplistic in-file access patterns: either sequential or random. With the emergence of large disk-image files that have complex internal structure, more accurate characterization of in-file access patterns becomes more important [51]. The techniques we use are well suited for creating benchmarks with such in-file workloads.

Our new virtual benchmarks are flexible and configurable, and support single- and multi-VM workloads. With multi-VM workloads, the emulated VMs can all run the same or *different* application workloads (a common feature of resource consolidation). Further, users do not need to go through a complex deployment process, such as hypervisor setup and per-VM OS and application installation, but can instead just run our benchmarks. Our benchmarks are invaluable for administrators that do not have access to the production environment when evaluating new or existing NAS servers for prospective virtualized clients. Finally, some benchmarks such as SPECsfs cannot be *usefully* run inside a VM because the benchmarks will continue to generate a physical workload to the NAS server; this means that new benchmarks can be the only viable evaluation option. Our benchmarks are capable of simulating a high load (i.e., many VMs) using only modest resources. Our experiments demonstrate that the accuracy of our benchmarks remains within 10% across 11 important parameters.

5.2 Background

In this section, we present several common data access methods for virtualized applications, describe in depth the changes in the virtualized NAS I/O stack (VM-NAS), and then explain the challenges in benchmarking NAS systems in virtualized environments.

5.2.1 Data Access Options for VMs

Many applications are designed to access data using a conventional POSIX file system interface. The methods that are currently used to provide this type of access in a VM can be classified into two categories: (1) emulated block devices (typically managed in the guest by a local file system); and (2) guest network file system clients.

Figure 5.1 illustrates both approaches. With an emulated block device, the hypervisor emulates an I/O controller with a connected disk drive. Emulation is completely transparent to the guest OS, and the virtual I/O controller and disk drives appear as physical devices to the OS. The guest OS typically formats the disk drive with a local file system or uses it as a raw block device. When an emulated block device is backed by file-based storage, we call the backing files *disk image files*.

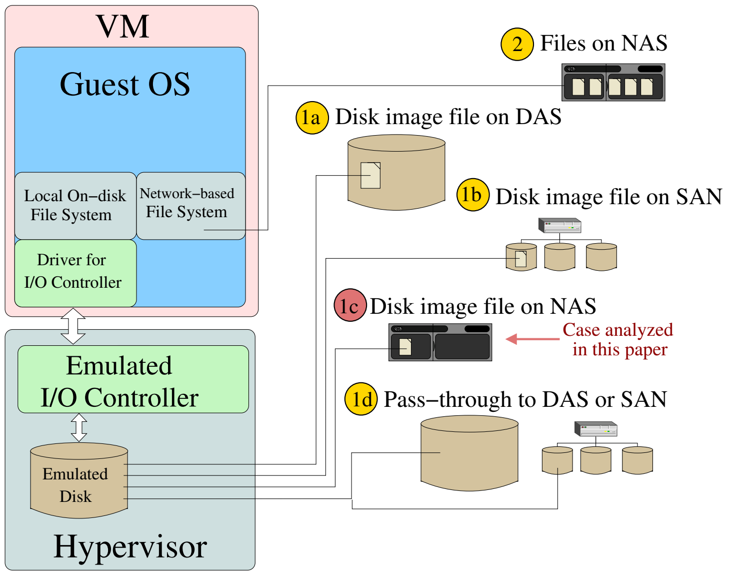

VM
Guest OS
Local On–disk File System
Network–based File System
Driver for I/O Controller
Emulated I/O Controller
Emulated Disk
Hypervisor
2 Files on NAS
1a Disk image file on DAS
1b Disk image file on SAN
1c Disk image file on NAS
Case analyzed in this paper
1d Pass–through to DAS or SAN

Figure 5.1: VM data-access methods. Cases 1a–1d correspond to the emulated-block-device architecture. Case 2 corresponds to the use of guest network file system clients.

Emulated Block Devices

Figure 5.1 shows several options for implementing the back end of an emulated block device:

1a. A file located on a local file system that is deployed on Direct Attached Storage (DAS). This approach is used, for example, by home and office installations of VMware Workstation [117] or Oracle VirtualBox [129]. Such systems often keep their disk images on local file systems (e.g., Ext3, NTFS). Although this architecture works for small deployments, it is rarely used in large enterprises where scalability, manageability, and high availability are critical.

1b. A disk image file is stored on a (possibly clustered) file system deployed over a Storage Area Network (SAN) (e.g., VMware's VMFS file system [131]). A SAN offers low-latency shared access to the available block devices, which allows high-performance clustered file systems to be deployed on top of the SAN. This architecture simplifies VM migration and offers higher scalability than DAS, but SAN hardware is more expensive and complex to administer.

1c. A disk image file stored on Network Attached Storage (NAS). In this architecture, which we call *VM-NAS*, the host's hypervisor passes I/O requests from the virtual machine to an NFS or SMB client, which in turn then accesses a disk image file stored on an external file server. The hypervisor is completely unaware of the storage architecture behind the NAS interface. NAS provides the scalability, reliability, and data mobility needed for efficient VM management. Typically, NAS solutions are cheaper than SANs due to their use of IP networks, and are simpler to configure and manage. These properties have increased the use of NAS in virtual environments and encouraged several companies to create solutions for disk image files management at the NAS [12, 111, 122].

1d. Pass-through to DAS or SAN. In this case, virtual disks are backed up by a real block device (not a file), which can be on a SAN or DAS. This approach is less flexible than disk image files, but can offer lower overhead because one level of indirection—the host file system—is eliminated.

Network Clients in the Guest

The other approach for providing storage to a virtual machine is to let a network-based file system (e.g., NFS) provide access to the data directly from the guest (case 2 in Figure 5.1). This model avoids the need for disk image files, so no block-device emulation is needed. This eliminates emulation overheads, but lacks many of the benefits associated with virtualization, such as consistent snapshots, thin provisioning, cloning, disaster recovery. Also, not every guest OS supports every NAS protocol, which fetters the ability of a hypervisor and its storage system to support all guest OS types. Further, cloud management architectures such as VMware's vCloud and OpenStack do not support this design [99, 127].

5.2.2 VM-NAS I/O Stack

In this study we focus on the VM-NAS architecture, where VM disks are emulated by disk image files stored on NAS (case 1c in Section 5.2.1 and in Figure 5.1). To the best of our knowledge, even though this architecture is becoming popular in virtual data centers [128, 143], there has been no study of the significant transformations in typical NAS I/O workloads caused by server virtualization. This study is a first step towards a better understanding of NAS workloads in virtualized environments and the development of suitable benchmarks for NAS to be used in industry and academia.

When VMs and NAS are used together, the corresponding I/O stack becomes deeper and more complex, as seen in Figure 5.2. When passing through the layers I/O requests significantly change their properties. At the top of the stack, applications access data using system calls such as `create`, `read`, `write`, and `unlink`. These system calls invoke the underlying guest file system, which in turn converts application calls into I/O requests to the block layer. The file system maintains data and metadata layouts, manages concurrent accesses, and often caches and prefetches data to improve application performance. All of these features change the pattern of application requests.

The guest OS's block layer receives requests from the file system and reorders and merges them to increase performance, provide process fairness, and prioritize requests. The I/O controller driver, located beneath the generic block layer, imposes extra limitations on the requests in accordance with the virtual device's capabilities (e.g., trims requests to the maximum supported size and limits the NCQ queue length [144]).

After that, requests cross the software-hardware boundary for the first time (here, the hardware is emulated). The hypervisor's emulated controller translates the guest's block-layer requests into reads and writes to the corresponding disk image files. Various request transformations can be done by the hypervisor to optimize performance and provide fair access to the data from multiple VMs [50].

The hypervisor contains its own network file system client (e.g., NFS), which can cache data, limit read and write sizes, and perform other request transformations. In this study we focus on NFSv3 because it is one of the most widely used protocols. However, our methodology is easily extensible to SMB or NFSv4. In the case of NFSv3, both the client and the server can limit read- and write-transfer sizes and modify write-synchronization properties. Because the hypervisor and its NFS client significantly change I/O requests, it is not sufficient to collect data at the block layer of the guest OS; we collect our traces at the entrance to the NFS server.

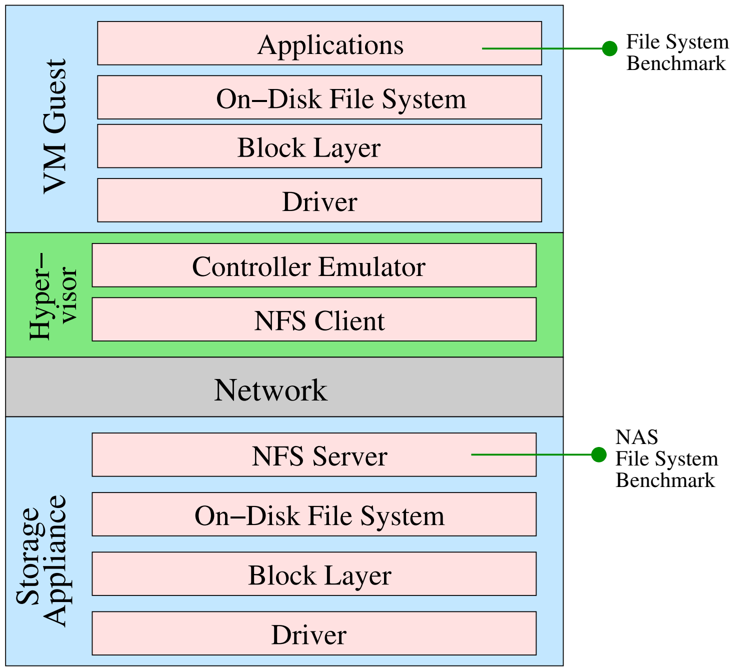


Figure 5.2: VM-NAS I/O Stack: VMs access and store virtual disk images on NAS.

After the request is sent over a network to the NAS server, the same layers that appear in the guest OS are repeated in the server. By this time, however, the original requests have already undergone significant changes performed by the upper layers, so the optimizations applied by similar layers at the server can be considerably different. Moreover, many NAS servers (e.g., NetApp [54]) run a proprietary OS that uses specialized request-handling algorithms, additionally complicating the overall system behavior. This complex behavior has a direct effect on measurement techniques, as we discuss next in Section 5.2.3.

5.2.3 VM-NAS Benchmarking Setup

Regular file system benchmarks usually operate at the application layer and generate workloads typical to one or a set of applications (Figure 5.2). In non-virtualized deployments these benchmarks can be used without any changes to evaluate the performance of a NAS server, simply by running the benchmark on a NAS client. In virtualized deployments, however, I/O requests can change significantly before reaching the NAS server due to the deep and diverse I/O stack described above. Therefore, benchmarking these environments is not straightforward.

One approach to benchmarking in a VM-NAS setup is to deploy the entire virtualization infrastructure and then run regular file system benchmarks inside the VMs. In this case, requests submitted by application-level benchmarks will naturally undergo the appropriate changes while passing through the virtualized I/O stack. However, this method requires a cumbersome setup of hypervisors, VMs, and applications. Every change to the test configuration, such as an increase in the number of VMs or a change of a guest OS, requires a significant amount of work. Moreover, the approach limits evaluation to the available test hardware, which may not be sufficient to run hypervisors with the hundreds of VMs that may be required to exercise the limits of the NAS server.

To avoid these limitations and regain the flexibility of standard benchmarks, we have created *virtualized benchmarks* by extracting the workload characteristics *after* the requests from the original *physical benchmarks* have passed though the virtualization and NFS layers. The generated benchmarks can then run directly against the NAS server without having to deploy a complex infrastructure. Therefore, the benchmarking procedure remains the same as before—easy, flexible, and accessible.

One approach to generating virtualized benchmarks would be to emulate the changes applied to each request as it goes down the layers. However, doing so would require a thorough study of the request-handling logic in the guest OSes and hypervisors, with further verification through multi-layer trace collection. Although this approach might be feasible, it is time-consuming, especially because it must be repeated for many different OSes and hypervisors. Therefore, in this work we chose to study the workload characteristics at a single layer, namely where requests enter the NAS server. We collected traces at this layer and then characterized selected workload properties. The information from a single layer is enough to create the corresponding NAS benchmarks by reproducing the extracted workload features. Workload characterization and the benchmarks that we create are tightly coupled with the configuration of the upper layers: application, guest OS, local file system, and hypervisor. We leave the inference of parameters that account for the greatest changes to the I/O workload for future work.

5.3 NAS Workload Changes

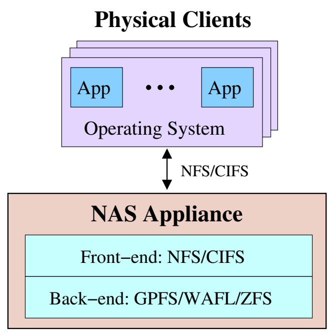
In this section we detail seven categories of NAS workload changes caused by virtualization. Specifically, we compare the two cases where a NAS server is accessed by a (1) *physical*; or (2) a *virtualized* client, and describe the differences in the I/O workload. These changes are the result of migrating an application from a physical server, which is configured to use an NFS client for direct data access, to a VM that stores data in a disk image file that the hypervisor accesses from an NFS server. Figure 5.3 demonstrates the difference in the two setups, and Table 5.2 summarizes the changes we observed in the I/O workload. The changes are listed from the most noticeable and significant to the least. Here, we discuss the changes qualitatively; quantitative observations are presented in Section 5.4.

First, and unsurprisingly, the number and size of files stored in NAS change from many relatively small files to a few (usually just one) large file(s) per VM—the disk image file(s). For example, the default Filebench file server workload defines 10,000 files with an average size of

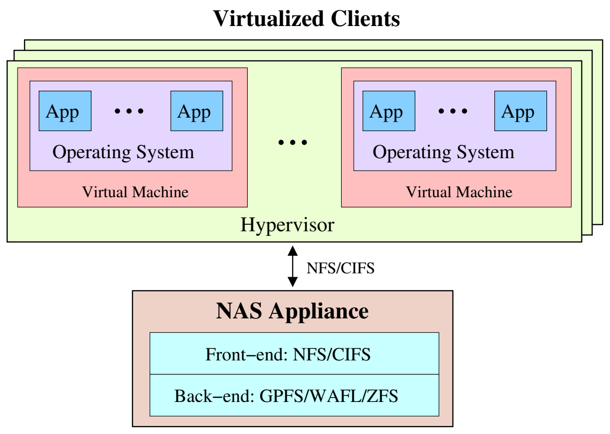
128KB, which are spread over 500 directories. However, when Filebench is executed in a VM, there is only one large disk image file. (Disk image files are usually sized to the space requirements of a particular application; in our setup the disk image file size was set to the default 16GB for the Linux VM, and to 50GB for the Windows VM, because the benchmark we used in Windows required at least 50GB.) For the same reason, directory depth decreases and becomes fairly consistent: VMware ESX typically has a flat namespace; each VM has one directory with the disk image files stored inside it. Back-end file systems used in NAS are often optimized for common file sizes and directory depths [4, 82, 92, 106], so this workload change can significantly affect their performance. For example, to improve write performance for small files, one popular technique is to store data in the inode [43], a feature that would be wasted on virtualized clients. Further, disk image files in NAS environments are typically sparse, with large portions of the files unallocated, i.e., the physical file size can be much smaller than its logical size. In fact, VMware's vSphere—the main tool for managing the VMs in VMware-based infrastructures—supports only the creation of sparse disk images over NFS. A major implication of this change is that back-end file systems for NAS can lower their focus on optimizing, for example, file append operations, and instead focus on improving the performance of block allocation within a file.

The second change caused by the move to virtualization is that all file system metadata operations become data operations. For example, with a physical client there is a one-to-one mapping between file creation and a CREATE over the wire. However, when the application creates a file in a VM, the NAS server receives a series of writes to a corresponding disk image: one to a directory block, one to an inode block, and possibly one or more to data blocks. Similarly, when an application accesses files and traverses the directory tree, physical clients send many LOOKUP procedures to a NAS server. The same application behavior in a VM produces a sequence of READs to the disk image. Current NAS benchmarks generate a high number of metadata operations (e.g., 72% for SPECsfs2008), and will bias the evaluation of a NAS that serves virtualized clients. While it may appear that removing all metadata operations implies that application benchmarks can generally be replaced with random I/O benchmarks, such as IOzone [24], this is insufficient.

As shown in Section 5.5, the VM-NAS I/O stack generates a range of I/O sizes, jump distances,



(a) Physical



(b) Virtualized

Figure 5.3: Physical and Virtualized NAS architectures. With physical clients, applications use a NAS client to access the NAS appliance directly. With virtualized clients, applications access the NAS appliance via a virtualized block device.

#	Workload Property	Physical NAS Clients	Virtual NAS Clients
1	File and directory count	Many files and directories	Single file per VM
	Directory tree depth	Often deeply nested directories	Shallow and uniform
	File size	Lean towards many small files	Multi-gigabyte sparse disk image files
2	Metadata operations	Many (72% in SPECsfs2008)	Almost none
3	I/O synchronization	Asynchronous and synchronous	All writes are synchronous
4	In-file randomness	Workload-dependent	Increased randomness due to guest file system encapsulation
	Cross-file randomness	Workload-dependent	Cross-file access replaced by in-file access due to disk image files
5	I/O Sizes	Workload-dependent	Increased or decreased due to guest file system fragmentation and I/O stack limitations
6	Read-modify-write	Infrequent	More frequent due to block layer in guest file system
7	Think time	Workload-dependent	Increased because of virtualization overheads

Table 5.2: Summary of key I/O workload changes between Physical and Virtualized NAS architectures.

and request offsets that cannot be modeled with a simple distribution (uniform or otherwise).

Third, all write requests that come to the NAS server are synchronous. For NFS, this means that the *stable* attribute is set on each and every write, which is typically not true for physical clients. The block layers of many OSes expect that when the hardware reports a write completion, the data has been saved to persistent storage. Similarly, the NFS protocol's stable attribute specifies that the NFS server cannot reply to a WRITE until the data is persistent. So the hypervisor satisfies the guest OS's expectation by always setting this attribute on WRITE requests. Since many modern NAS servers try to improve performance by gathering write requests into larger chunks in RAM, setting the stable attribute invalidates this important optimization for virtualized clients.

Fourth, in-file randomness increases significantly with virtualized clients. On a physical client, access patterns (whether sequential or random) are distinct on a per-file basis. However, in virtualized clients, both sequential and random operations are blended into a single disk image file. This causes the NAS server to receive what appears to be many more random reads and writes to that file. Furthermore, guest file system fragmentation increases image file randomness. On the other hand, cross-file randomness decreases, as each disk image file is typically accessed by only a single VM; i.e., it can be easier to predict which files will be accessed next based on files' status, and to differentiate files by how actively the files are used (running VMs, stopped ones, etc.).

Fifth, the I/O sizes of original requests can both decrease and increase while passing through the virtualization layers. Guest file systems perform reads and writes in units of their block size, often 4KB. So, when reading a file of, say, 6KB size, the NAS server observes two 4KB reads for a total of 8KB, while a physical client would request only 6KB (25% less). Since many modern systems operate with a lot of small files [92], this difference can have a significant impact on bandwidth. Similarly, when reading 2KB of data from two consecutive data blocks in a file (1KB in each block), the NAS server may observe two 4KB reads for a total of 8KB (one for each block), while a physical NAS client may send only a single 2KB request. A NAS server designed for a virtualized environment could optimize its block-allocation and fragmentation-prevention strategies to take advantage of this observation.

Interestingly, I/O sizes can also *decrease* because guest file systems sometimes split large files into blocks that might not be adjacent. This is especially true for aged file systems with higher fragmentation [112]. Consequently, whereas a physical client might pass an application's 1MB read directly to the NAS, a virtualized client can sometimes submit several smaller reads scattered across the (aged) disk image. An emulated disk controller driver can also reduce the size of an I/O request. For example, we observed that the Linux IDE driver has a maximum I/O size of 128KB, which means that any application requests larger than this value will be split into smaller chunks. Note that such workload changes happen even in a physical machine as requests flow from a file system to a physical disk. However, in a VM-NAS setup, the transformed requests hit not a real disk, but a file on NAS, and as a result the NAS experiences a different workload.

The sixth change is that when an application writes to part of a block, the guest file system must perform a read-modify-write (RMW) to first read in valid data prior to updating and writing it back to the NAS server. Consequently, virtualized clients often cause RMWs to appear on the wire [52], requiring two block-sized round trips for every update. With physical clients, the RMW is generally performed at the NAS server, avoiding the need to first send valid data back to the NAS client.

Seventh, the think time between I/O requests can increase due to varying virtualization overhead. It has been shown that for a single VM and modern hardware, the overhead of virtualization is small [7]. However, as the number of VMs increases, the contention for computational resources grows, which can cause a significant increase in the request inter-arrival times. Longer think times can prevent a NAS device from filling the underlying hardware I/O queues and achieving peak throughput.

In summary, both static and dynamic properties of NAS workloads change when virtualized clients are introduced into the infrastructure. The changes are sufficiently significant that direct comparison of certain workload properties between virtual and physical clients becomes problematic. For example, cross-file randomness has a rather different meaning in the virtual client, where the number of files is usually one per VM. Therefore, in the rest of the chapter we focus solely on characterizing workloads from virtualized clients, without trying to compare them directly against the physical clients. However, where possible, we refer to the original workload properties.

5.4 VM-NAS Workload Characterization

In this section we describe our experimental setup and then present and characterize a set of four different application-level benchmarks.

5.4.1 Experimental Configuration

Every layer in the VM-NAS I/O stack can be configured in several ways: different guest OSes can be installed, various virtualization solutions can be used, etc. The way in which the I/O stack is assembled and configured can significantly change the resulting workload. In the current work we did not try to evaluate every possible configuration, but rather selected several representative setups to demonstrate the utility of our techniques. The methodology we have developed is simple

Parameter	**RHEL 6.2**	**Win 2008 R2 SP1**
No. of CPUs	1	
Memory	1GB	2GB
Host Controller	Paravirtual	LSI Logic Parallel
Disk Drive Size	16GB	50GB
Disk Image Format	Thick flat VMDK	
Guest File System	Ext3	NTFS
Guest I/O Scheduler	CFQ	n/a

Table 5.3: Virtual Machine configuration parameters.

and accessible enough to evaluate many other configurations. Table 5.3 presents the key configuration options and parameters we used in our experiments. Since our final goal is to create NAS benchmarks, we only care about the settings of the layers above the NAS server; we treat the NAS itself as a black box.

We used two physical machines in our experimental setup. The first acted as a *NAS server*, while the second represented a typical *virtualized client* (see Figure 5.3). The hypervisor was installed on a Dell PowerEdge R710 node with an Intel Xeon E5530 2.4GHz 4-core CPU and 24GB of RAM. We used local disk drives in this machine for the hypervisor installation—VMware ESXi 5.0.0 build 62386. We used two guest OSes in the virtual setup: Red Hat Enterprise Linux 6.2 (RHEL 6.2) and Windows 2008 R2 SP1. We stored the OS's VM disk images on the local, directly attached disk drives. We conducted our experiments with a separate virtual disk in every VM, with the corresponding disk images being stored on the NAS. We pre-allocated all of the disk images (thick provisioning) to avoid performance anomalies across runs related to thin provisioning (e.g., delayed block allocations). The RHEL 6.2 distribution comes with a paravirtualized driver for VMware's emulated controller, so we used this controller for the Linux VM. We left the default format and mount options for guest file systems unchanged.

The machine designated as the NAS server was a Dell PowerEdge 1800 with six 250GB Maxtor 7L250S0 disk drives connected through a Dell CERC SATA 1.5/6ch controller, intended to be used as a storage server in enterprise environments. The machine is equipped with an Intel Xeon 2.80GHz Irwindale single-core CPU and 512MB of memory. The NAS server consisted of both the Linux NFS server and IBM's General Parallel File System (GPFS) version 3.5 [109]. GPFS is a scalable clustered file system that enables a scale-out, highly-available NAS solution and is used in both virtual and non-virtual environments. Our workload characterization and benchmark synthesis techniques treat NAS servers as a black box and are valid regardless of its underlying hardware and software. Since our ultimate goal is to create benchmarks capable of stressing any NAS, we did not characterize NAS-specific characteristics such as request latencies. Our benchmarks, however, let us manually configure the think time. By decreasing think time (along with increasing the number of VMs), a user can scale the load to the processing power of a NAS to accurately measure its peak performance.

5.4.2 Application-Level Benchmarks

In the Linux VM we used Filebench [39] to generate file system workloads. Filebench can emulate the I/O patterns of several enterprise applications; we used the File-, Web-, and Database-server

Workload	**Dataset size**	**Files**	**R/W/M ratio**	**I/O Size**
File-server	2.0GB	20,000	1/2/3	WF
Web-server	1.6GB	100,000	10/1/0	WF
DB-server	2.0GB	10	10/1/0	2KB
Mail-server	24.0GB	120	1/2/0	32KB

Table 5.4: High-level workload characterization for our benchmarks. R/W/M is the Read/Write/-Modify ratio. WF (Whole-File) means the workload only reads or writes complete files. The mail-server workload is based on JetStress, for which R/W/M ratios and I/O sizes were estimated based on [64].

workloads. We scaled up the datasets of these workloads so that the datasets were larger than the amount of RAM in the VM (see Table 5.4).

Because Filebench does not support Windows, in our Windows VM we used JetStress 2010 [63], a disk-subsystem benchmark that generates a Microsoft Exchange Mail-server workload. It emulates accesses to the Exchange database by a specific number of users, with a corresponding number of log file updates. Complete workload configurations (physical and virtualized), along with all the software we developed as part of this project are available from `https://avatar.fsl.cs.sunysb.edu/groups/t2mpublic/`.

Although SPECsfs is a widely used NAS benchmark [116], we could not use it in our evaluation because it incorporates its own NFS client, which makes it impossible to run against a regular POSIX interface. We hope that the workload analysis and proposed benchmarks presented in this study can be used by SPEC for designing future SPECsfs synthetic workloads.

VMware's VMmark is a benchmark often associated with testing VMs [130]. However, this benchmark is designed to evaluate the performance of a hypervisor machine, not the underlying storage system. For example, VMmark is sensitive to how fast a hypervisor's CPU is and how well it supports virtualization features (such as AMD-V and Intel VT [1, 60]). However, these details of hypervisor configuration should not have a large effect on NAS benchmark results. Although VMmark also indirectly benchmarks the I/O subsystem, it is hard to distinguish how much the I/O component contributes to the overall system performance. Moreover, VMmark requires the installation of several hypervisors and additional software (e.g., Microsoft Exchange) to generate the load. Our goal is complementary: to design a realistic benchmark for the NAS that serves as the backend storage for a hypervisor like VMware.

Our goal in this project was to transform some of the already existing benchmarks to their virtualized counterparts. As such, we did not replay any real-world traces in the VMs. Both Filebench and JetStress generate workloads whose statistical characteristics remain the same over time (i.e., stationary workloads). Consequently, new virtualized benchmarks also exhibit this property.

5.4.3 Characterization

We executed all benchmarks for 10 minutes (excluding the preparation phase) and collected NFS traces at the NAS server. We repeated every run 3 times and verified the consistency of the results. The traces were collected using the GPFS *mmtrace* facility [59] and then converted to the DataSeries format [9] for efficient analysis.

We developed a set of tools for extracting various workload characteristics. There is always a nearly infinite number of characteristics that can be extracted from a trace, but a NAS benchmark needs to reproduce only those that significantly impact the performance of NAS servers. Since there is no complete list of workload characteristics that impact NAS, in the future we plan to conduct a systematic study of NASes to create such a list. For this study, we selected characteristics that clearly affect most NASes: (1) read/write ratio; (2) I/O size; (3) jump (seek) distance; and (4) offset popularity.

As we mentioned earlier, the workloads produced by VMs contain no metadata operations. Thus, we only characterize the ratio of data operations—READs to WRITEs. The jump distance of a request is defined as the difference in offsets (block addresses) between it and the immediately preceding request (accounting for I/O size as well). We do not take the operation type into account when calculating the jump distance. The offset popularity is a histogram of the number of accesses to each block within the disk image file; we report this as the number of blocks that were accessed once, twice, etc. We present the offset popularity and I/O size distributions on a per-operation basis. Figure 5.4 depicts the read/write ratios and Figures 5.5–5.8 present I/O size, jump distance, and offset popularity distributions for all workloads. For jump distance we show a CDF because it is the clearest way to present this parameter.

Read/Write ratio. Read/write ratios vary significantly across the analyzed workloads. The File-server workload generates approximately the same number of reads and writes, although the original workload had twice as many writes (Table 5.4). We attribute this difference to the high number of metadata operations (e.g., LOOKUPs and STATs) that were translated to reads by the I/O stack. The Web-server and the Database-server are read-intensive workloads, which is true for both original and virtualized workloads. The corresponding original workloads do not contain many metadata operations, and therefore the read/write ratio remained unchanged (unlike the File-server workload). The Mail-server workload, on the other hand, is write-intensive: about 70% of all operations are writes, which is close to the original benchmark where two thirds of all operations are writes. As with the Web-server and Database-server workloads, the lack of metadata operations kept the read/write ratio unchanged,

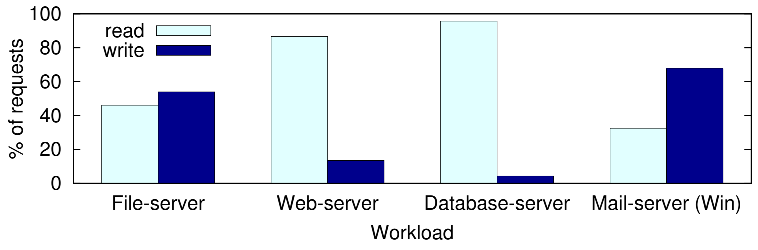
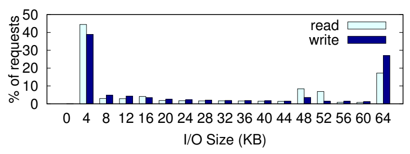
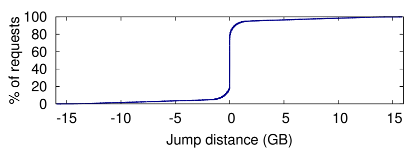


Figure 5.4: Read/Write ratios for different workloads.

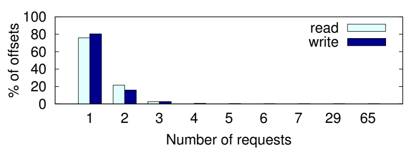
I/O size distribution. The I/O sizes for all workloads vary from 512B to 64KB; the latter limit is imposed by the RHEL 6.2 NFS server, which sets 64KB as the default maximum NFS read and write size. All requests smaller than 4KB correspond to 0 on the bar graphs. There are few writes smaller than 4KB for the File-server and Web-server workloads, but for the Database- and Mail-server (JetStress) workloads the corresponding percentages are 80% and 40%, respectively. Such small writes are typical for databases (Microsoft Exchange emulated by JetStress also uses a database) for two reasons. First, the Database-server workload writes 2KB at a time using direct I/O. In this case, the OS page cache is bypassed during write handling, and consequently the I/O



(a) I/O Size

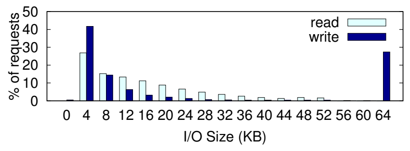


(b) Jump distance

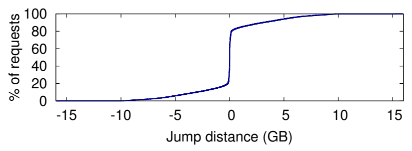


(c) Offset popularity

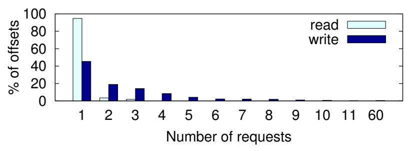
Figure 5.5: Characteristics of a virtualized File-server workload.



(a) I/O Size

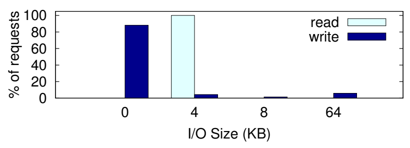


(b) Jump distance

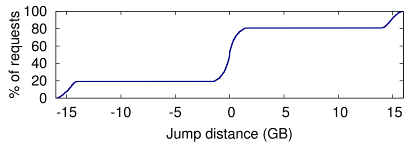


(c) Offset popularity

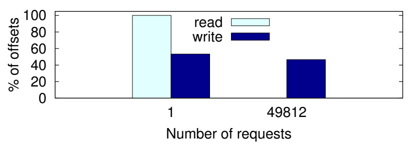
Figure 5.6: Characteristics of a virtualized Web-server workload.



(a) I/O Size

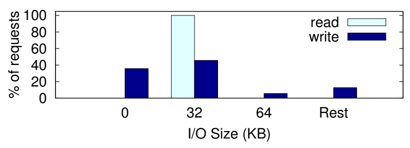


(b) Jump distance

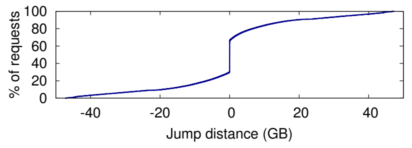


(c) Offset popularity

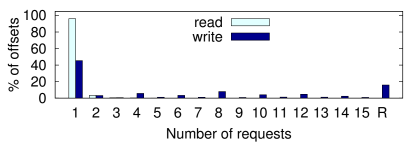
Figure 5.7: Characteristics of a virtualized Database-server workload.



(a) I/O Size



(b) Jump distance



(c) Offset popularity

Figure 5.8: Characteristics of a virtualized Mail-server workload.

size is not increased to 4KB (the page size) when it reaches the block layer. The block layer cannot then merge requests, due to their randomness. Second, databases often perform operations synchronously by using the `fsync` and `sync` calls. This causes the guest file system to atomically update its metadata, which can only be achieved by writing a single sector (512B) to the virtual disk drive (and hence over NFS).

For the File-server and Web-server workloads, most of the writes happen in 4KB and 64KB I/O sizes. The 4KB read size is dominant in all workloads because this is the guest file system block size. However, many of the File-server's reads were merged into larger requests by the I/O scheduler and then later split into 64KB sizes by the NFS client. This happens because the average file size for the File-server is 128KB, so whole-file reads can be merged. For the Web-server workload, the average file size is only 16KB, so there are no 64KB reads at all. For the same reason, the Web-server workload exhibits many reads around 16KB (some files are slightly smaller, others are slightly larger, in accordance with Filebench's gamma distribution [137]). Interestingly, for the Mail-server workload, many requests have non-common I/O sizes. (We define an I/O size as non-common if fewer than 1% of such requests have such I/O size.) We grouped all non-common I/O sizes in the bucket called "Rest" in the histogram. This illustrates that approximately 15% of all requests have non-common I/O sizes for the Mail-server workload.

Jump distance. The CDF jump distance distribution graphs show that many workloads demonstrate a significant level of sequentiality, which is especially true for the File-server workload: more than 60% of requests are sequential. Another 30% of the requests in the File-server workload represent comparatively short jumps: less than 2GB, the size of the dataset for this workload; these are jumps between different files in the active dataset. The remaining 10% of the jumps come from metadata updates and queries, and are spread across the entire disk. The Web-server workload exhibits similar behavior except that the active dataset is larger—about 5–10GB. The cause of this is a larger number of files in the workload (compared to File-server) and the allocation policy of Ext3 that tries to spread many files across different block groups.

For the Database-server workload there are almost no sequential accesses. Over 60% of the jumps are within 2GB because that is the dataset size. Interestingly, about 40% of the requests have fairly long jumps that are caused by frequent file system synchronization, which leads to metadata updates at the beginning of the disk.

In the Mail-server workload approximately 40% of the requests are sequential, and the rest are spread across the 50GB disk image file. A slight bend around 24GB corresponds to the active dataset size. Also, note that the Mail-server workload uses the NTFS file system, which uses a different allocation policy than Ext3; this explains the difference in the shape of the Mail-server curve from other workloads.

Offset popularity. In all workloads, most of the offsets were accessed only once. The absolute numbers on these graphs depend on the run time, e.g., when one runs a benchmark longer, then the chance of accessing the same offset increases. However, the shape of the curve remains the same as time progresses (although it shifts to the right). For the Database workload, 40% of all blocks were updated several thousand times. We attribute this to the repeated updates of the same file system metadata structures due to frequent file system synchronization. The Mail-server workload demonstrates a high number of overwrites (about 50%). These overwrites are caused by Microsoft Exchange overwriting the log file multiple times. With Mail-server, “R” on the X axes designates the “Rest” of the values, because there were too many to list. We therefore grouped all of the values that contributed less than 1% into the R bucket.

5.5 New NAS Benchmarks

This section describes our methodology for the creation of new NAS benchmarks for virtualized environments and then evaluates their accuracy.

5.5.1 Trace-to-Model Conversion

Our NAS benchmarks generate workloads with characteristics that closely follow the statistical distributions presented in Section 5.4.3. We decided not to write a new benchmarking tool, but rather exploit Filebench’s ability to express I/O workloads with its Workload Modeling Language (WML) [138], which allows one to flexibly define processes and performed I/O operations. Filebench interprets WML and translates its instructions to corresponding POSIX system calls. Our use of Filebench will facilitate the adoption of our new virtualized benchmarks: existing Filebench users can easily run new WML configurations.

We extended the WML language to support two virtualization terms: *hypervisor* and *vm* (virtual machine). We call the extended version WML-V (by analogy with AMD-V). WML-V is backwards compatible with the original WML, so users can merge virtualized and non-virtualized configurations to simultaneously emulate the workloads generated by both physical and virtual clients.

For each analyzed workload—File-server, Web-server, Database-server and Mail-server—we created a corresponding WML-V configuration file. By modifying these files, a user can adjust the workloads to reflect a desired benchmarking scenario, e.g., defining the number of VMs and the workloads each VM runs.

Listing 5.1 presents an abridged example of a WML-V configuration file that defines a single hypervisor, which runs 5 Database VMs and 2 Web-server VMs. *Flowops* are Filebench's defined I/O operations, which are mapped to POSIX calls, such as `open`, `create`, `read`, `write`, and `delete`. In the VM case, we only use read and write flowops, since metadata operations do not appear in the virtualized workloads. For every defined VM, Filebench will pre-allocate a disk image file of a user-defined size—16GB in the example listing.

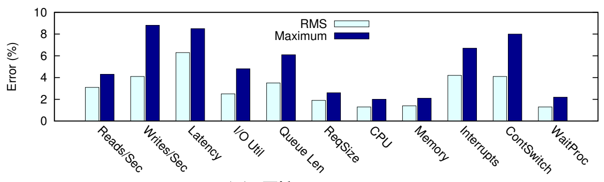
```
HYPERVISOR name="physical-host1" {
 VM name="dbserver-vm",dsize=16gb,instances=5 {
  flowop1, ...
 }
 VM name="websever-vm",dsize=16gb,instances=2 {
  flowop1, ...
 }
}
```

Listing 5.1: An abridged WML-V workload description that defines 7 VMs: 5 run database workloads and 2 generate Web-server workloads.

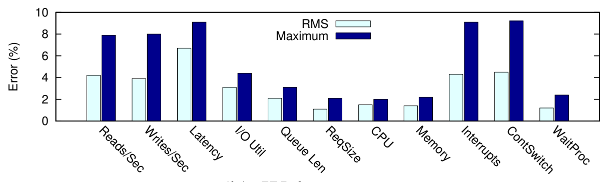
Filebench allows one to define random variables with desired empirical distributions; various flowop attributes can then be assigned to these random variables. We used this ability to define read and write I/O-size distributions and jump distances. We achieved the required read/write ratios by putting an appropriate number of read and write flowops within the VM definition. The generation of a workload with user-defined jump distances and offset popularity distributions is a complex problem [80] that Filebench does not solve; in this work, we do not attempt to emulate this parameter. However, as we show in the following section, this does not significantly affect the accuracy of our benchmarks.

Ideally, we would like Filebench to translate flowops directly to NFS procedures. However, this would require us to implement an NFS client within Filebench (which is an ongoing effort within the Filebench community). To work around this limitation, we mount NFS with the `sync` flag and `open` the disk image files with the O_DIRECT flag, ensuring that I/O requests bypass the Linux page cache. These settings also ensure that (1) no additional read requests are performed to the NFS server (readahead); (2) that all write requests are immediately sent to the NFS server without modification; and (3) that replies are returned only after the data is on disk. This behavior was validated with extensive testing. This approach works well in this scenario because we do not need to generate metadata procedures on the wire; that would be difficult to achieve using this method because a 1:1 mapping of metadata operations does not exist between system calls and NFS procedures.

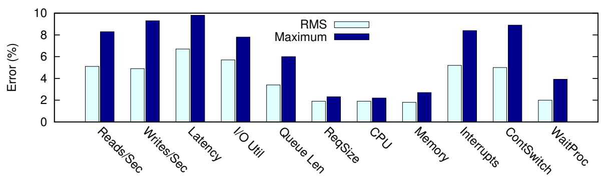
Our enhanced Filebench reports aggregate operations per second for all VMs and individually for each VM. Operations in the case of virtualized benchmarks are different from the original



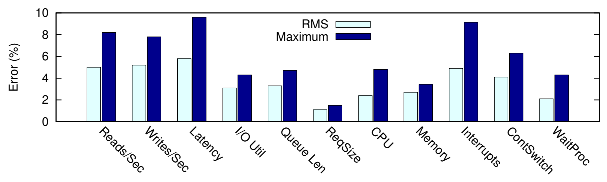
(a) File-server



(b) Web-server



(c) Database-server



(d) Mail-server (Windows, JetStress)

Figure 5.9: Root Mean Square (RMS) and maximum relative distances of response parameters for all workloads.

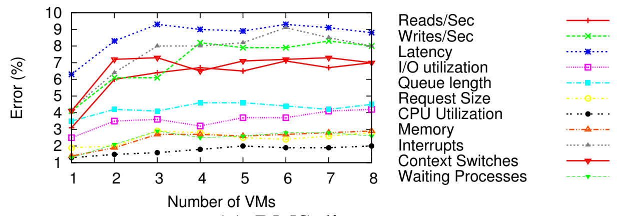
non-virtualized equivalent: our benchmarks report the number of reads and writes per second; application-level benchmarks, however, report application-level operations (e.g., the number of HTTP requests serviced by a Web-server). Nevertheless, the numbers reported by our benchmarks can be directly used to compare the performance of different NAS servers under a configured workload.

None of our original benchmarks, except the database workload, emulated think time, because our test was designed as an I/O benchmark. For the database benchmark we defined think time as originally defined in Filebench—200,000 loop iterations. Think time in all workloads can be adjusted by trivial changes to the workload description.

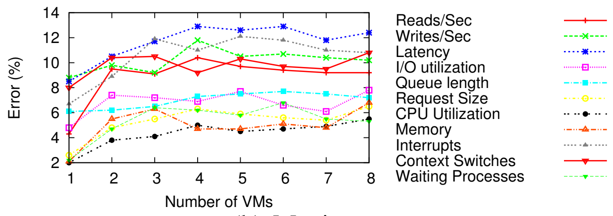
5.5.2 Evaluation

To evaluate the accuracy of our benchmarks we observed how the NAS server responds to the virtualized benchmarks as compared to the original benchmarks when executed in a VM. We monitored 11 parameters that represent the response of a NAS and are easy to extract through the Linux */proc* interface: (1) Reads/second from the underlying block device; (2) Writes/second; (3) Request latency; (4) I/O utilization; (5) I/O queue length; (6) Request size; (7) CPU utilization; (8) Memory usage; (9) Interrupt count; (10) Context-switch count; and (11) Number of processes in the wait state. We call these *NAS response parameters*.

We sampled the response parameters every 30 seconds during a 10-minute run and calculated the relative difference between each pair of parameters. Figure 5.9 presents maximum and Root Mean Square (RMS) difference we observed for four workloads. In these experiments a single VM with an appropriate workload was used. The maximum relative error of our benchmarks is always less than 10%, and the RMS distance is within 7% across all parameters. Certain response parameters show especially high accuracy; for example, the RMS distance for request size is within 4%. Here, the accuracy is high because our benchmarks directly emulate I/O size distribution. Errors in CPU and memory utilization were less than 5%, because the NAS in our experiments did not perform many CPU-intensive tasks.



(a) RMS distance



(b) Maximum

Figure 5.10: Response parameter errors depending on the number of VMs deployed. The first four VMs (1–4) execute four different workloads we analyzed. The next four VMs (5–8) are repeated in the same order.

Scalability with Multiple Virtual Machines. The benefit of our benchmarks is that a user can define many VMs with different workloads and measure NAS performance against this specific workload configuration. To verify that the accuracy of our benchmarks does not decrease as we emulate more VMs, we conducted a multi-VM experiment. We first ran one VM with a File-server in it, then added a second VM with a Web-server workload, then a third VM executing the Database-server workload, and finally a fourth VM running JetStress. After that we added another four VMs with the same four workloads in the same order. In total we had 8 different configurations ranging from 1 to 8 VMs; this setup was designed to heavily stress the NAS under several, different, concurrently running workloads. We then emulated the same 8 configurations using our benchmarks and again monitored the response parameters. Figures 5.10(a) and 5.10(b) depict RMS and maximum relative errors, respectively, depending on the number of VMs.

When a single VM is emulated, our benchmarks show the best accuracy. Beyond one VM, the RMS error increased by about 3–5%, but still remained within 10%. For four parameters—latency, writes/sec, interrupts and context switches count—the maximum error observed during the whole run was the highest among other parameters—in the 10–13% range.

In summary, our benchmarks show a high accuracy for both single- and multi-VM experiments, even under heavy stress.

5.6 Related Work

Storage performance in virtualized environments is an active research area. Le et al. studied the storage performance implications of combining different guest and host file systems [81]. Boutcher et al. examined how the selection of guest OS and host I/O schedulers impacts the performance of a virtual machine [20]. Both of these works focused on the performance aspects of the problem, not workload characterization or generation; also, the authors used direct-attached storage, which is simpler but less common in modern enterprise data centers.

Hildebrand et al. discussed the implications of using the VM-NAS architecture with enterprise storage servers [52]. That work focused on the performance implications of the VM-NAS I/O stack without thoroughly investigating the changes to the I/O workload. Gulati et al. characterized the SAN workloads produced by VMs for several enterprise applications [49]. Our techniques can also be used to generate new benchmarks for SAN-based deployments, but we selected to investigate VM-NAS setups first, for two reasons. First, NAS servers are becoming a more popular solution for hosting VM disk images. Second, the degree of workload change in such deployments is higher: NAS servers use more complex network file-system protocols whereas SANs and DAS use a simpler block-based protocol.

Ahmad et al. studied performance overheads caused by I/O stack virtualization in ESX with a SAN [7]. That study did not focus on workload characterization but rather tried to validate that modern VMs introduce low overhead compared to physical nodes. Later, the same authors proposed a low-overhead method for on-line workload characterization in ESX [6]. However, their tool characterizes traces collected at the virtual SCSI layer and consequently does not account for any transformations that may occur in ESX and its NFS client. In contrast, we collect the trace at the NAS layer after all request transformations, allowing us to create more accurate benchmarks.

Casale et al. proposed a model for predicting storage performance when multiple VMs use shared storage [25, 74]. Practical benchmarks like ours are complementary to that work and allow one to verify such predictions in real life. Ben-Yehuda et al. analyzed performance bottlenecks when several VMs are used to provide different functionalities on a storage controller [14]. The authors focused on lowering network overhead via intelligent polling and other techniques.

Trace-driven performance evaluation and workload characterization have been the basis of many studies [37, 70, 77, 101]. Our trace-characterizing techniques and benchmark-synthesis techniques are based on multi-dimensional workload analysis. Chen et al. used multi-dimensional trace analysis to infer behavior of enterprise storage systems [29]. Tarasov et al. proposed a technique for automated translation of block-I/O traces to workload models [121]. Yadawakar et al. proposed to discover applications based on multi-dimensional characteristics of NFS traces [142].

In summary, to the best of our knowledge, there have been no earlier studies that systematically analyzed virtualized NAS workloads. Moreover, we are the first to present new NAS benchmarks that accurately generate virtualized I/O workloads.

5.7 Conclusions

We have studied the transformation of existing NAS I/O workloads due to server virtualization. Such transformations were known to occur but have not been studied in depth to date. Our analysis revealed several significant I/O workload changes due to the use of disk images and the placement of the guest block layer above the NAS file client. We observed and quantified significant changes such as the disappearance of file system metadata operations at the NAS layer, changes in I/O sizes, changes in file counts and directory depths, asynchrony changes, increased randomness within files, and more.

Based on these observations from real-world workloads, we developed new benchmarks that accurately represent NAS workloads in virtualized data centers—and yet these benchmarks can be run directly against the NAS without requiring a complex virtualization environment configured with VMs and applications. Our new virtualized benchmarks represent four workloads, two guest operating systems, and up to eight virtual machines. Our evaluation reveals that the relative error of these new benchmarks across more than 11 parameters is less than 10% on average. In addition to providing a directly usable measurement tool, we hope that our work will provide guidance to future NAS standards, such as SPEC, in devising benchmarks that are better suited to virtualized environments.

Chapter 6

Conclusion

Workloads play a crucial role in designing and optimizing modern storage systems. In fact, when designing a system the majority of decisions—starting from the selection of a file system block size and ending with the deduplication algorithm—are based on the properties of the target workload.

As the gap between the performance of storage components and the amount of stored data widens the need for workload-based optimizations will only increase. Practical and efficient tools for characterizing real workloads and their synthesis are needed to address this issue.

In this thesis we demonstrated the problems of evaluating complex storage systems and proposed Multi-Dimensional Histogram (MDH) technique for analyzing workloads and system behaviors. Using three examples we showed the effectiveness of MDH technique in evaluating a variety of workload-driven optimizations.

First, we applied MDH technique for converting I/O traces to workload models. Our workload model consists of a sequence of MDHs that preserve important workload features. To address the variability of workload properties in the trace, we perform trace chunking. Further, we eliminate chunks that exhibit similar workload properties to reduce the trace model's size. Our evaluation demonstrates that the accuracy of generated models approaches 95%, while the model size is less than 6% of the original trace size. Such concise models enable easy comparison, scaling, and other modifications.

Second, we used MDH to generate realistic datasets suitable for evaluating deduplication systems. Our generic framework emulates file system data and metadata changes, which we call mutations. Our implementation of the mutation module for the framework captures the statistics of changes observed across several real datasets using MDH and a Markov Model. The model demonstrates low error rate—less than 15% for 71 mutations across all datasets.

Third, we characterized how the workloads experienced by NAS servers change when the servers are accessed by virtualized clients. We observed and quantified significant changes such as the disappearance of file system metadata operations at the NAS layer, changes in I/O sizes, changes in file counts and directory depths, asynchrony changes, increased randomness within files, and more. Using MDH technique we created a set of versatile benchmarks that generate virtualized workloads without deploying complex infrastructure. Our evaluation reveals that the relative error of these new benchmarks across more than 11 parameters is less than 10% on average.

MDH-based techniques are versatile and powerful for workload analysis and synthesis. It is our hope that the contributions presented here will benefit research and engineering communities.

6.1 Future Work

We see at least three promising research directions related to MDH. First, workload models created using MDH, unlike workload traces and snapshots, are mathematical objects. Investigating the operations on these objects is an interesting research thrust. Especially appealing looks the study of tools and techniques that can scale the MDH along one or several dimensions. This will allow performance engineers to sensibly adjust workload features to match new expected workloads. Other tools can combine two or more workload models so that the resulting model represents several consolidated applications. In addition, tools for comparing various MDHs are of significant interest. They form the basis for identifying the classes of similar real world workloads.

Second, our experience asserts that visualizing MDH for further analysis is a complex but extremely useful task. Observing workload changes in the traces and performing human-assisted chunking are some of the important use cases for MDH visualization. It is for the future researchers to apply existing techniques on visualizing multi-dimensional space to MDH in the context of storage evaluation [139].

Third, when many MDHs are collected from different environments, clustering techniques can be applied to detect similar workloads. This will allow to identify workload classes common in the real world and guide the development of the future systems.

In addition to the three generic MDH research directions mentioned above, there are studies specific to three application areas presented in this thesis. They are described below.

Trace to Workload Model Conversion. We used block traces when building our trace to model converter. As file system interface remains popular in the modern deployments, supporting the file system traces is a valuable feature. File system traces contain an operation field (READ, WRITE, STAT, CREATE, etc.) and the arguments of the operation depend on the specific operation. Studying such type of traces can introduce certain changes to the MDH technique and should be thoroughly evaluated.

Analysis of the traces collected from multiple layers in the I/O stack allows to find important correlations between I/O layers and create more accurate workload models. Because MDH is a universal technique we believe it is the right choice to be applied across many layers.

Our current chunking method is simple and investigating alternative chunking techniques is an interesting research direction. In fact, Talwadker and Voruganti have recently presented an alternative chunking technique that avoids fixed chunking during the initial stage in the trace to model conversion [118].

In this work we used existing benchmarks to generate workloads. However, creating a new benchmark that takes MDH as an input allows more accurate workload generation. Such a benchmark will not have the limitations caused by the low expressiveness of the existing benchmarks.

Realistic Dataset Generation. Our specific implementation of the framework modules might not model all parameters that potentially impact the behavior of existing deduplication systems. We believe that a study similar to Park et al. [104] should be conducted to create a complete list of the dataset properties that impact deduplication systems.

Although we can generate an initial file system snapshot using a specially collected profile for FS-MUTATE, such approach can be limiting. In future, an extensive study on how to create initial fstree objects can be performed.

Many deduplication systems perform local chunk compression to achieve even higher aggregate compression. Developing a method for generating chunks with a realistic compression ratio is consequently a useful extension.

It is also interesting to investigate whether one can use extended traces of user and application I/O activity to emulate file system evolution more accurately. Our system is mostly suitable for evaluating backup deduplications systems but inline deduplication systems require emulating dynamic properties of the traces. We believe that MDH suits well for solving this problem because it preserves dependencies between the dimensions.

NAS Workloads in Virtualized Setups. The number of NAS benchmarks should be extended by analyzing actual applications and application traces, including typical VM operations such as booting, updating, and snapshotting—and examine root and I/O swap partition access patterns. We also believe that exploring more VM configuration options such as additional guest file systems (and their age), hypervisors, and NAS protocols is an important research direction.

Once a larger body of virtual NAS benchmarks exists, the research community will be able to study the I/O workload's sensitivity to each configuration parameter as well as investigate the impact of extracting and reproducing additional trace characteristics in the generated benchmarks.

At the moment, benchmark creation requires manual analysis for every application the need to be emulated. In the future, one can investigate the feasibility of automatic transformation of physical workloads to virtual workloads via a multi-level trace analysis of the VM-NAS I/O stack.

Bibliography

[1] Advanced Micro Devices, Inc. Industry leading virtualization platform efficiency, 2008. `www.amd.com/virtualization`. §5.4.2

[2] N. Agrawal, A. C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 125–138, San Francisco, CA, February 2009. USENIX Association. §4.3.1, §4.3.3, §4.5.1, §4.7

[3] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 31–45, San Jose, CA, February 2007. USENIX Association. §4.5.1, §4.7

[4] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 3–3, San Jose, CA, February 2007. USENIX Association. §5.3

[5] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-level security for network-attached disks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 159–174, San Francisco, CA, March 2003. USENIX Association. §2.1

[6] I. Ahmad. Easy and efficient disk I/O workload characterization in VMware ESX server. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2007. §5.6

[7] I. Ahmad, J. M. Anderson, A. M. Holler, R. Kambo, and V. Makhija. An analysis of disk performance in VMware ESX server virtual machines. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2003. §5.3, §5.6

[8] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 139–152, San Francisco, CA, February 2009. USENIX Association. §2.8, §3.1, §3.6

[9] E. Anderson, M. Arlitt, C. Morrey, and A. Veitch. DataSeries: an efficient, flexible, data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1), January 2009. §3.4, §5.4.3

[10] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58, San Francisco, CA, March/April 2004. USENIX Association. §2.8, §3.6

[11] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: a file system to trace them all. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association. §2.6, §2.8, §3.6

[12] Atlantis Computing. `www.atlantiscomputing.com/`. §5.2.1

[13] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40:33–37, December 2007. §2.4

[14] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and B. Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association. §5.6

[15] J. M. Bennett, M. A. Bauer, and D. Kinchlea. Characteristics of files in NFS environments. *ACM SIGSMALL/PC Notes*, 18(3-4):18–25, 1992. §4.7

[16] Thomas Bittman. *Virtual machines and market share through 2012*. Gartner, October 2009. ID Number: G00170437. §5.1

[17] Thomas Bittman. *Q&A: six misconceptions about server virtualization*. Gartner, July 2010. ID Number: G00201551. §5.1

[18] P. Bodik, A. Fox, M. Franklin, M. Jordan, and D. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the First ACM Symposium on Cloud Computing (SOCC)*, pages 241–252, 2010. §2.8, §3.6

[19] R. E. Bohn and J. E. Short. How much information? 2009 report on american consumers. `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf`, December 2009. §4.1

[20] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '09)*, October 2009. §5.6

[21] T. Bray. The Bonnie home page. `www.textuality.com/bonnie`, 1996. §2.2

[22] Alan D. Brunelle. Blktrace user guide, February 2007. §2.7

[23] M. Burtscher. Tcgen 2.0: a tool to automatically generate lossless trace compressors. *ACM SIGARCH Computer Architecture News*, 34:1–8, 2006. §3.5

[24] D. Capps. IOzone file system benchmark. `www.iozone.org`. §2.2, §3.1, §3.2, §5.3

[25] G. Casale, S. Kraft, and D. Krishnamurthy. A model of storage I/O performance interference in virtualized systems. In *Proceedings of the International Workshop on Data Center Performance (DCPerf)*, 2011. §5.6

[26] M. Chamness. Capacity forecasting in a backup storage environment. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2011. §4.7

[27] P. M. Chen and D. 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 International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, Seattle, WA, May 1993. ACM SIGOPS. §2.3.1

[28] Y. Chen, S. Alspaugh, and R. Katz. Interactive query processing in big data systems: A cross-industry study of mapreduce workloads. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB '12)*, Istanbul, Turkey, August 2012. Morgan Kaufmann. §1

[29] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press. §2.4, §3.6, §5.6

[30] R. Coker. The Bonnie++ home page. `www.coker.com.au/bonnie++`, 2001. §2.2

[31] EMC Corporation. EMC Centra: content addressed storage systems. Product description guide, 2004. §4.1

[32] A. Coskun, R. Strong, D. Tullsen, and T. S. Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the 2009 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2009. §3.5

[33] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2011. USENIX Association. §4.1, §4.4

[34] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*. ACM, 1999. §4.7

[35] A. B. Downey. The structural cause of file size distributions. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, 2001. §4.7

[36] M. Ebling and M. Satyanarayanan. SynRGen: An extensible file reference generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994. ACM. §2.8, §3.2

[37] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Everything you always wanted to know about NFS trace analysis, but were afraid to ask. Technical Report TR-06-02, Harvard University, Cambridge, MA, June 2002. §5.6

[38] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003. USENIX Association. §3.2

[39] Filebench. `http://filebench.sf.net`. §2.1, §2.2, §2.3, §3.1, §3.2, §4.3.2, §4.3.3, §4.7, §5.4.2

[40] fio—flexible I/O tester. `http://freshmeat.net/projects/fio/`. §2.2

[41] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proceedings of the International Workshop on Information and Software as Services (WISS)*, 2010. §3.6

[42] G. Ganger. Generating representative synthetic workloads: an unsolved problem. In *Proceedings of Computer Measurement Group Conference (CMG)*, 1995. §3.6

[43] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, January 1997. USENIX Association. §5.3

[44] John Gantz and David Reinsel. Extracting value from chaos. IDC 1142, June 2011. §1

[45] GNU. GSL - GNU Scientific Library, 2013. `http://www.gnu.org/software/gsl/`. §3.3.3

[46] M. Gomez and V. Santonja. A new approach in the modeling and generation of synthetic workloads. In *Proceedings of the 8th Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000. §3.6

[47] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller. Self-similarity in File Systems. In *Proceedings of ACM SIGMetrics/Performance*, 1998. §3.1, §3.5

[48] Advanced Storage Products Group. Identifying the hidden risk of data deduplication: how the HYDRAstor solution proactively solves the problem. Technical Report WP103-3_0709, NEC Corporation of America, 2009. §4.1

[49] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Proceedings of 2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009. §5.6

[50] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman. Demand based hierarchical QoS using storage resource pools. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, June 2012. USENIX Association. §5.2.2

[51] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP '11)*, Cascais, Portugal, October 2011. ACM Press. §2.6, §5.1

[52] D. Hildebrand, A. Povzner, R. Tewari, and V. Tarasov. Revisiting the storage stack in virtualized NAS environments. In *Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2011. §5.3, §5.6

[53] E. Hille and R. Phillips. Functional analysis and semi-groups. *AMS Colloquium Publications*, 31:300–327, 1957. §3.3.3

[54] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994. USENIX Association. §5.2.2

[55] B. Hong and T. Madhyastha. The relevance of long-range dependence in disk traffic and implications for trace synthesis. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2005. §3.6

[56] B. Hong, T. Madhyastha, and B. Zhang. Cluster-based input/output trace analysis. In *Proceedings of 24th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2005. §3.6

[57] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988. §2.2

[58] IBM. IBM scale out metwork attached storage. `www.ibm.com/systems/storage/network/sonas/`. §3.5

[59] IBM. General Parallel File System problem determination guide. Technical Report GA22-7969-02, IBM, December 2004. `http://pic.dhe.ibm.com/infocenter/db2luw/v9r8/index.jsp?topic=%2Fcom.ibm.db2.luw.sd.doc%2Fdoc%2Ft0056934.html`. §5.4.3

[60] Intel Corporation. Intel virtualization technology (Intel VT), 2008. `www.intel.com/technology/virtualization/`. §5.4.2

[61] Iometer. `http://iometer.sourceforge.net`, August 2003. §2.2

[62] N. Jain, M. Dahlin, and R. Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 281–294, San Francisco, CA, December 2005. USENIX Association. §4.1

[63] Microsoft Exchange Server JetStress 2010. `www.microsoft.com/en-us/download/details.aspx?id=4167`. §5.4.2

[64] Understanding database and log performance factors. http://technet.microsoft.com/en-us/library/ee832791.aspx. §5.4

[65] K. Jin and E. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM. §4.1, §4.7

[66] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS. §2.3.2

[67] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 337–350, San Francisco, CA, December 2005. USENIX Association. §2.6, §2.8, §3.1, §3.6

[68] J. Katcher. PostMark: a new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html. §2.2, §2.4, §2.7

[69] S. Kavalanekar, D. Narayanan, S. Sankar, E. Thereska, K. Vaid, and B. Worthington. Measuring database performance in online services: a trace-based approach. In *Proceedings of TPC Technology Conference on Performance Evaluation and Benchmarking (TPC TC)*, 2009. §3.1

[70] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2008. §2.8, §3.5, §3.6, §5.6

[71] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI 1996)*, pages 19–34, Seattle, WA, October 1996. §3.1, §3.6

[72] A. Konwinski, J. Bent, J. Nunez, and M. Quist. Towards an I/O tracing framework taxonomy. In *In Proceedings of the International Workshop on Petascale Data Storage (PDSW)*, 2007. §2.8, §3.6

[73] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM. §4.5.1

[74] S. Kraft, G. Casale, D. Krishnamurthy, D. Greer, and P. Kilpatrick. Performance models of storage contention in cloud environments. *Software and Systems Modeling*, 12(2), March 2012. §5.6

[75] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 105–118, Boston, MA, June 2001. USENIX Association. §2.1

[76] G. Kuenning. Mersenne Twist Pseudorandom Number Generator Package, 2010. `http://lasr.cs.ucla.edu/geoff/mtwist.html`. §3.3.2

[77] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer 1994 USENIX Conference*, pages 291–303, June 1994. §2.8, §3.1, §3.6, §4.7, §5.6

[78] Z. Kurmas. *Generating and Analyzing Synthetic Workloads using Iterative Distillation*. PhD thesis, Georgia Institute of Technology, 2004. §3.2

[79] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing representative I/O workloads using iterative distillation. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, 2003. §2.8, §3.6

[80] Z. Kurmas, J. Zito, L. Trevino, and R. Lush. Generating a jump distance based synthetic disk access pattern. In *Proceedings of the International IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2006. §5.5.1

[81] D. Le, H. Huang, and H. Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association. §5.6

[82] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC '08)*, pages 213–226, Berkeley, CA, 2008. USENIX Association. §4.7, §5.3

[83] T. Li and L. K. John. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 160–171, 2003. §3.5

[84] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 173–186, San Francisco, CA, March/April 2004. USENIX Association. §3.2

[85] A. Liguori and E. Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization (WIOV '08)*, 2008. §4.7

[86] LLNL. LLNL IOR: I/O Performance Benchmark. `https://asc.llnl.gov/sequoia/benchmarks/#ior`. §2.2

[87] Wei Lu, Jared Jackson, and Roger Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM. §2.2

[88] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, pages 431–441, 1963. §3.3.3

[89] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, pages 3–30, 1998. §3.3.2

[90] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM. §4.7

[91] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //TRACE: parallel trace replay with approximate causal events. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 153–167, San Jose, CA, February 2007. USENIX Association. §2.8, §3.6

[92] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–1, San Jose, CA, February 2011. USENIX Association. §4.1, §4.1, §4.7, §5.3, §5.3

[93] Changwoo Mina, Kangnyeon Kimb, Hyunjin Choc, Sang-Won Leed, and Young Ik Eome. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association. §1

[94] M. Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics*, 1, January 2002. §4.7

[95] J. Mogul. Brittle metrics in operating systems research. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 90–95, Rio Rica, AZ, March 1999. ACM. §2.8

[96] R. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, June 2001. §3.6

[97] Omar Al-Ubaydli Neal S. Young, John P. A. Ionnadis. Why current publication practices may distort science. *PLoS Med*, 5, October 2008. `www.plosmedicine.org/article/info:doi/10.1371/journal.pmed.0050201`. §2.1

[98] NetApp. NetApp deduplication for FAS. Deployment and implementation, 4th revision. Technical Report TR-3505, NetApp, 2008. §4.1

[99] OpenStack Foundation. `www.openstack.org/`. §5.2.1

[100] OSDL. Iometer project. `www.iometer.org`. §2.2, §3.1, §3.2

[101] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles (SOSP)*, pages 15–24, Orcas Island, WA, December 1985. ACM. §2.8, §3.1, §3.6, §4.7, §5.6

[102] Y. Pan, G. Dong, and T. Zhang. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011. §3.5

[103] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010. §4.1, §4.7

[104] N. Park, W. Xiao, K. Choi, and D. J. Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmark. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011. §6.1

[105] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A Balanced Energy-Efficiency Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Beijing, China, June 2007. §2.2

[106] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association. §4.7, §5.3

[107] J. Santos and S. Rao. `http://sourceforge.net/projects/ffsb/`. §2.2

[108] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP '81)*, pages 15–24. ACM Press, 1981. §4.7

[109] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association. §3.5, §5.4.1

[110] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Ferformance and Energy in File System Server Workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association. §3.5

[111] Simplivity. `www.simplivity.com/`. §5.2.1

[112] K. A. Smith and M. I. Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1997)*, pages 203–213. ACM, 1997. §4.7, §5.3

[113] SNIA. SNIA - storage networking industry association: IOTTA repository, 2007. `http://iotta.snia.org`. §2.6

[114] Storage Networking Industry Association (SNIA). Block I/O trace common semantics (working draft). `www.snia.org/sites/default/files/BlockIOSemantics-v1.0r11.pdf`, February 2010. §3.4

[115] SPEC. SPEC SFS97 (2.0) benchmark. `www.spec.org/osg/sfs97`, June 2001. §2.2

[116] SPEC. SPECsfs2008. `www.spec.org/sfs2008`, July 2008. §5.1, §5.4.2

[117] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstations hosted virtual machine monitor. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, Boston, MA, June 2001. USENIX Association. §5.2.1

[118] Rukma Talwadker and Kaladhar Voruganti. Paragone: Whats next in block I/O trace modeling. In *Proceedings of the International IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, Nevada, May 2010. IEEE. §6.1

[119] D. Tang and M. Seltzer. Lies, damned lies, and file system benchmarks. Technical Report TR-34-94, Harvard University, December 1994. In VINO: The 1994 Fall Harvest. §2.1, §2.8

[120] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011. §2.8

[121] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association. §2.8, §5.6

[122] Tintri. `www.tintri.com/`. §5.2.1

[123] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008. §2.1, §2.8, §3.5

[124] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice*. Society for Industrial and Applied Mathematics, 2012. §3.3, §3.3.3

[125] A. Tridgell. dbench-3.03 README. `http://samba.org/ftp/tridge/dbench/README`, 1999. §2.2

[126] UMass trace repository. `http://traces.cs.umass.edu`. §3.5

[127] VMware vCloud. `http://vcloud.vmware.com/`. §5.2.1

[128] Richard Villars and Noemi Greyzdorf. *Worldwide file-based storage 2010–2014 forecast update*. IDC, December 2010. IDC #226267. §5.1, §5.2.2

[129] VirtualBox. `https://www.virtualbox.org/`. §5.2.1

[130] VMMark. `www.vmware.com/go/vmmark`. §5.4.2

[131] VMware, Inc. *VMware Virtual Machine File System: Technical Overview and Best Practices*, 2007. `www.vmware.com/pdf/vmfs-best-practices-wp.pdf`. §5.2.1

[132] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Charleston, SC, December 1999. ACM. §4.7

[133] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association. §4.7

[134] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal behavior of real traffic data. In *Proceedings of Performance*, 2002. §3.6

[135] M. Wang, T. Madhyastha, N. Chan, and S. Papadimitriou. Data mining meets performance evaluation: fast algorithms for modeling burst traffic. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*, 2002. §3.6

[136] Watts up? PRO ES Power Meter. `www.wattsupmeters.com/secure/products.php`. §3.5

[137] A. W. Wilson. Operation and implementation of random variables in Filebench. §5.4.3

[138] Filebench Workload Model Language (WML). `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench_Workload_Language`. §5.5.1

[139] Pak Chung Wong and Daniel Bergeron. 30 years of multidimensional multivariate visualization. *Scientific Visualization, Overviews, Methodologies, and Techniques. Washington, DC, USA: IEEE Computer Society*, pages 3–33, 1997. §6.1

[140] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 239–252, San Francisco, CA, February 2009. USENIX Association. §3.5

[141] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011. §4.1

[142] N. Yadwadkar, C. Bhattacharyya, and K. Gopinath. Discovery of application workloads from network file traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, San Jose, CA, February 2010. USENIX Association. §2.8, §3.6, §5.6

[143] Natalya Yezhkova, Liz Conner, Richard L. Villars, and Benjamin Woo. *Worldwide enterprise storage systems 2010–2014 forecast: recovery, efficiency, and digitization shaping customer requirements for storage systems*. IDC, May 2010. IDC #223234. §5.1, §5.2.2

[144] Y. Yu, D. Shin, H. Eom, and H. Yeom. NCQ vs I/O scheduler: preventing unexpected misbehaviors. *ACM Transaction on Storage*, 6(1), March 2010. §5.2.2

[145] Erez Zadok and Geoff Kuenning. Benchmarking and Tracing: New Horizons. `https://www.usenix.org/legacy/events/fast11/bofs.html#zadok`, February 2011. §2.8

[146] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing representative I/O workloads for TPC-H. In *Proceedings of International Sypmposium on High Performance Computer Architecture (HPCA)*, 2004. §3.6

[147] Z. Zhang and K. Ghose. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 59–72, San Francisco, CA, March 2003. USENIX Association. §2.1

[148] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, USA, 2008. §4.5.1

[149] N. Zhu, J. Chen, and T. Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 323–336, San Francisco, CA, December 2005. USENIX Association. §2.5, §2.6, §2.6, §2.8, §3.6

[150] N. Zhu, J. Chen, T. Chiueh, and D. Ellard. An NFS trace player for file system evaluation. Technical Report TR-14-03, Harvard University, December 2003. §3.6