A Nine Year Study of File System and Storage Benchmarking

AVISHAY TRAEGER and EREZ ZADOK
Stony Brook University
and
NIKOLAI JOUKOV and CHARLES P. WRIGHT
IBM T. J. Watson Research Center

Benchmarking is critical when evaluating performance, but is especially difficult for file and storage systems. Complex interactions between I/O devices, caches, kernel daemons, and other OS components result in behavior that is rather difficult to analyze. Moreover, systems have different features and optimizations, so no single benchmark is always suitable. The large variety of workloads that these systems experience in the real world also adds to this difficulty.

In this article we survey 415 file system and storage benchmarks from 106 recent papers. We found that most popular benchmarks are flawed and many research papers do not provide a clear indication of true performance. We provide guidelines that we hope will improve future performance evaluations. To show how some widely used benchmarks can conceal or overemphasize overheads, we conducted a set of experiments. As a specific example, slowing down read operations on ext2 by a factor of 32 resulted in only a 2–5% wall-clock slowdown in a popular compile benchmark. Finally, we discuss future work to improve file system and storage benchmarking.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*Measurements*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques; performance attributes*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Benchmarks, file systems, storage systems

ACM Reference Format:

Traeger, A., Zadok, E., Joukov, N., and Wright, C. P. 2008. A nine year study of file system and storage benchmarking. ACM Trans. Storage 4, 2, Article 5 (May 2008), 56 pages. DOI = 10.1145/1367829.1367831 http://doi.acm.org/ 10.1145/1367829.1367831

This work was made possible partially thanks to NSF awards CNS-0133589 (CAREER), CCR-0310493 (CyberTrust), CNS-0614784 (CSR), and CCF-0621463 (HECURA), and well as two HP/Intel gifts numbers 87128 and 88415.1.

Authors' addresses: A. Traeger (corresponding author), E. Zadok, Computer Science Department, Stony Brook University, Stony Brook, NY 11794; email: atraeger@cs.sunysb.edu; N. Joukov, C. P. Wright, IBM TJ Watson Research Center, PO Box 704, Yorktown Heights, NY 10598.

1. INTRODUCTION

Benchmarks are most often used to provide an idea of how fast some piece of software or hardware runs. The results can significantly add to, or detract from, the value of a product (be it monetary or otherwise). For example, they may be used by potential consumers in purchasing decisions, or by researchers to help determine a system's worth.

When a performance evaluation of a system is presented, the results and implications must be clear to the reader. This includes accurate depictions of behavior under realistic workloads and in worst-case scenarios, as well as explaining the reasoning behind benchmarking methodologies. In addition, the reader should be able to verify the benchmark results, and compare the performance of one system with that of another. To accomplish these goals, much thought must go into choosing suitable benchmarks and configurations, and accurate results must be conveyed.

Ideally, users could test performance in their own settings using real workloads. This transfers the responsibility of benchmarking from author to user. However, this is usually impractical because testing multiple systems is time consuming, especially in that exposing the system to real workloads implies learning how to configure the system properly, possibly migrating data and other settings to the new systems, as well as dealing with their respective bugs. In addition, many systems developed for research purposes are not released to the public. Although rare, we have seen performance measured using actual workloads when they are created for in-house use [Ghemawat et al. 2003] or are made by a company to be deployed [Schmuck and Haskin 2002; Eisler et al. 2007]. Hence, the next best alternative is for some party (usually the authors) to run workloads that are representative of real-world use on commodity hardware. These workloads come in the form of synthetic benchmarks, executing real programs, or using traces of some activity. Simulating workloads raises concerns about how accurately these benchmarks portray the end-user's workload. Thus, benchmarks must be well understood so as to not have unknown side-effects, and should provide a good approximation of how the program would perform under different loads.

Benchmarking file and storage systems requires extra care, which exacerbates the situation. Even though these systems share the goal of providing access to data via a uniform API, they differ in many ways, such as type of underlying media (e.g., magnetic disk, network storage, CD-ROM, volatile RAM, flash RAM, etc.), storage environment (e.g., RAID, LVM, virtualization, etc.), the workloads for which the system is optimized, and in their features (e.g., journals, encryption, etc.).

In addition, complex interactions exist between file systems, I/O devices, specialized caches (e.g, buffer cache, disk cache), kernel daemons (e.g., `kflushd`), and other OS components. Some operations may be performed asynchronously, and this activity is not always captured in benchmark results. Because of this complexity, many factors must be taken into account when performing benchmarks and analyzing the results.

In this article we concentrate on file and storage system benchmarks in the research community. Specifically, we comment on how to choose and create benchmarks, how to run them, and how to analyze and report the results. We have surveyed a selection of recent file and storage system papers and have found several poor benchmarking practices, as well as some good ones. We classify the benchmarks into three categories and discuss them in turn. The categories are as follows.

—*Macrobenchmarks*. The performance is tested against a particular workload that is meant to represent some real-world workload.
—*Trace Replays*. A program replays operations which were recorded in a real scenario, with the hope that it is representative of real-world workloads.
—*Microbenchmarks*. A few (typically one or two) operations are tested to isolate their specific overheads within the system.

The rest of this article is organized as follows. In Section 2 we describe the criteria for selecting publications for our survey and list the papers we analyze. Section 3 provides suggested guidelines to use when benchmarking and Section 4 discusses how well the surveyed papers have followed those suggestions. In Section 5 we give an overview of related research. We describe the system configuration- and benchmarking procedures that were used in our experiments in Section 6.

Section 7 reviews the pros and cons of the macrobenchmarks used in the surveyed papers; we also include a few other notable benchmarks for completeness. In Section 8 we examine how the papers that we surveyed used traces, and we describe four main problems that arise when using traces for performance analysis. Section 9 describes the widely used microbenchmarks, and Section 10 discusses some of the more popular workload generators.

Section 11 describes a suite of tools for benchmarking automation. Section 12 shows the benchmarks that we performed. We conclude in Section 13, summarizing our suggestions for choosing the proper benchmark, and offering our ideas for the future of file and storage system benchmarking.

2. SURVEYED PAPERS

Research papers have used a variety of benchmarks to analyze the performance of file and storage systems. This article surveys the benchmarks and benchmarking practices from a selection of the following recent conferences.

—the Symposium on Operating Systems Principles (SOSP 1999, 2001, 2003, and 2005);
—the Symposium on Operating Systems Design and Implementation (OSDI 2000, 2002, 2004, and 2006);
—the USENIX Conference on File and Storage Technologies (FAST 2002, 2003, 2004, 2005, and 2007); and
—the USENIX Annual Technical Conference (USENIX 1999, 2000, 2001, 2002, 2003, 2004, 2005, and 2006).

Research papers relating to file systems and storage often appear in the proceedings of these conferences, which are considered to be of high quality. We decided to consider only full-length papers from conferences that have run for at least five years. In addition, we consider only file systems and storage papers that have evaluated their implementations (no simulations), and from those, only papers whose benchmarks are used to measure performance in terms of latency or throughput. For example, precluded from our work are those papers whose benchmarks were used to verify correctness or report on the amount of disk space used. Studies similar to ours have been performed in the past [Small et al. 1997; Mogul 1999], and so we believe that this cross-section of conference papers is adequate to make some generalizations. We surveyed 106 papers in total, 8 of which are our own. The surveyed papers are marked with asterisks in the References section.

3. RECOMMENDED BENCHMARKING GUIDELINES

We now present a list of guidelines to consider when evaluating the performance of a file or storage system. A Web-version summary of this document can be found at www.fsl.cs.sunysb.edu/project-fsbench.html.

The two underlying themes of these guidelines are the following.

(1) *Explain What Was Done in as Much Detail as Possible*. For example, if one decides to create one's own benchmark, the paper should detail what was done. If replaying traces, one should describe where they came from, how they were captured, and how they were replayed (what tool? what speed?). This can help others understand and validate the results.
(2) *In Addition to Saying What Was Done, Say Why It Was Done That Way*. For example, while it is important to note that one is using ext2 as a baseline for the analysis, it is just as important (or perhaps even moreso) to discuss why it is a fair comparison. Similarly, it is useful for readers to know why one ran that random-read benchmark so that they know what conclusions to draw from the results.

3.1 Choosing The Benchmark Configurations

The first step of evaluating a system is to pose questions that will reveal the performance characteristics of the system, such as "how does my system compare to current similar systems?", "how does my system behave under its expected workload?", and "what are the causes of my performance improvements or overheads?" Once these questions are formulated, one must decide on what baseline systems, system configurations, and benchmarks should be used to best answer them. This will produce a set of ⟨system, configuration, benchmark⟩ tuples that will need to be run. It is desirable for the researcher to have a rough idea of the possible expected results for each configuration at this point; if the actual results differ from these expectations, then the causes of the deviations should be investigated.

Since a system's performance is generally more meaningful when compared to the performance of existing technology, one should find existing systems that provide fair and interesting comparisons. For example, for benchmarking an encryption storage device, it would be useful to compare its performance to those of other encrypted storage devices, a traditional device, and perhaps some alternate implementations (user-space, file system, etc.).

The system under test may have several configurations that will need to be evaluated in turn. In addition, one may create artificial configurations where a component of the system is removed to determine its overhead. For example, in an encryption file or storage system, one can use a null cipher (copy data only), rather than encryption, to isolate the overhead of encryption. Determining the cause of overheads may also be done using profiling techniques. Showing this incremental breakdown of performance numbers helps the reader to better understand a system's behavior.

There are three main types of benchmark that one can choose from: macrobenchmarks, trace replaying, and microbenchmarks.

—*Macrobenchmarks*. These exercise multiple file system operations, and are usually good for an overall view of the system's performance, though the workload may not be realistic. These benchmarks are described further in Section 7.

—*Trace-Based*. Replaying traces can also provide an overall view of the system's performance. Traces are usually meant to exercise the system with a representative real-world workload, which can help to better understand how a system would behave under normal use. However, one should ensure that the trace is in fact representative of that workload (e.g., the trace should capture a large enough sample), and that the method used to replay the trace preserves the characteristics of the workload. Section 8 provides more information about trace-based benchmarking.

—*Microbenchmarks*. These exercise few (usually one or two) operations. These are useful if one is measuring a very small change to better understand the results of a macrobenchmark, to isolate the effects of specific parts of the system, or to show worst-case behavior. In general, these benchmarks are more meaningful when presented together with other benchmarks. See Section 9 for more information.

Useful file system benchmarks should highlight the high-level as well as low-level performance. Therefore, we recommend using at least one macrobenchmark or trace to show a high-level view of performance, along with several microbenchmarks to highlight more focused views. In addition, there are several workload properties that might be considered. We describe here five which we believe are important. First, benchmarks may be characterized by how CPU or I/O bound they are. File and storage system benchmarks should generally be I/O bound, but a CPU-bound benchmark may also be run for systems that exercise the CPU. Second, if the benchmark records its own timings, it should use accurate measurements. Third, the benchmark should be scalable, meaning that it exercises each machine the same amount, independent of hardware or software speed. Fourth, multithreaded workloads may provide more realistic scenarios, and may help to saturate the system with requests. Fifth, the workloads should be well understood. Although the code of synthetic benchmarks can be read and traces analyzed, it is more difficult to understand some application workloads.

For example, compile benchmarks can behave rather differently depending on the testbed's architecture, installed software, and the version of software being compiled. The source-code for ad hoc benchmarks should be publicly released, as it is the only truly complete description of the benchmark that would allow others to reproduce it (including any bugs or unexpected behavior).

3.2 Choosing The Benchmarking Environment

The state of the system during the benchmark's runs can have a significant effect on results. After determining an appropriate state, it should be created accurately and reported along with the results. Some major factors that can affect results are cache state, ZCAV effects, file system aging, and nonessential processes running during the benchmark.

The state of the system's caches can affect the code paths that are tested and thus affect benchmark results. It is not always clear whether benchmarks should be run with "warm" or "cold" caches. On one hand, real systems do not generally run with completely cold caches. On the other hand, a benchmark that accesses too much cached data may be unrealistic as well. Because requests are mainly serviced from memory, the file or storage system will not be adequately exercised. Further, not bringing the cache back to a consistent state between runs can cause timing inconsistencies. If cold-cache results are desired, caches should be cleared before each run. This can be done by allocating and freeing large amounts of memory, remounting the file system, reloading the storage driver, or rebooting. We have found that rebooting is more effective than the other methods [Wright et al. 2005]. When working in an environment with multiple machines, the caches on all necessary machines must be cleared. This helps create identical runs, thus ensuring more stable results. If, however, warm-cache results are desired, this can be achieved by running the experiment N+1 times, and discarding the first run's result.

Most modern disks use zoned constant angular velocity (ZCAV) to store data. In this design, the cylinders are divided into zones where the number of sectors in a cylinder increases with the distance from the center of the disk. Therefore the transfer rate varies from zone to zone [Van Meter 1997]. It has been recommended to minimize ZCAV effects by creating a partition of the smallest possible size on the outside of the disk [Ellard and Seltzer 2003b]. However, this makes the results less realistic and may not be appropriate for all benchmarks (e.g., long seeks may be necessary to show the effectiveness of the system). We recommend simply specifying the location of the test partition in the paper, so as to help reproducibility.

Most file system and storage benchmarks are run on an empty system, which could make the results different than in a real-world setting. A system may be aged by running a workload based on system snapshots [Smith and Seltzer 1997]. However, aging a 1GB file system by seven months using this method required writing 87.3GB of data. The amount of time required to age a file system would make this impractical for larger systems. TBBT has a faster, configurable aging technique but is somewhat less realistic, as it is purely synthetic [Zhu et al. 2005a]. Other methods to age a system before running a benchmark are to run a long-term workload, copy an existing raw image, or to replay a trace before running the benchmark. It should be noted that for some systems and benchmarks, aging is not a concern. For example, aging will not have any effect when replaying a block-level trace on a traditional storage device, since the benchmark will behave identically regardless of the disk's contents.

To ensure reproducibility of the results, all nonessential services and processes should be stopped before running the benchmark. These processes can cause anomalous results (outliers) or higher-than-normal standard deviations for a set of runs. However, processes such as `cron` will coexist with the system when used in the real world, and so it must be understood that these results are measured in a sterile environment. Ideally, we would be able to demonstrate performance with the interactions of other processes present; however, this is difficult because the set of processes is specific to a machine's configuration. Instead, we recommend using multithreaded workloads because they more accurately depict a real system, which normally has several active processes. In addition, we recommend to ensure that no users log into the test machines during a benchmark run, and to also ensure that no other traffic is consuming one's network bandwidth while running benchmarks that involve the network.

3.3 Running The Benchmarks

We recommend four important guidelines to running benchmarks properly. First, one should ensure that every benchmark run is identical. Second, each test should be run several times to ensure accuracy, and standard deviations or confidence levels should be computed to determine the appropriate number of runs. Third, tests should be run for a period of time sufficient for the system to reach steady state for the majority of the run. Fourth, the benchmarking process should preferably be automated using scripts or available tools such as Auto-pilot [Wright et al. 2005] to minimize the mistakes associated with manual repetitive tasks. This is discussed further in Section 11.

3.4 Presenting The Results

Once results are obtained, they should be presented appropriately so that accurate conclusions may be derived. Aside from the presented data, the benchmark the configurations and environment should be accurately described. Proper graphs should be displayed, with error bars, where applicable.

We recommend using confidence intervals rather than standard deviation to present results. The standard deviation is a measure of the amount of variation between runs. The half-width of the confidence interval describes how far the true value may be from the captured mean with a given degree of confidence (e.g., 95%). This provides a better sense of the true mean. In addition, as more benchmark runs are performed, the standard deviation may not decrease, but the width of confidence intervals generally will.

For experiments with fewer than 30 runs, one should be careful not to use the normal distribution for calculating confidence intervals. This is because the central-limit theorem no longer holds with a small sample size. Instead, one should use the student's t-distribution. This distribution may also be used for experiments with at least 30 runs, since in this case it is similar to the normal distribution.

Large confidence-interval widths or nonnormal distributions may indicate a software bug or benchmarking error. For example, the half-widths of confidence intervals are recommended to be less than 5% of the mean. If the results are not stable, then either there is a bug in the code or the instability should be explained. Anomalous results (e.g., outliers) should never be discarded. If they are due to programming or benchmarking errors, the problem should be fixed and the benchmarks rerun to gather new and more stable results.

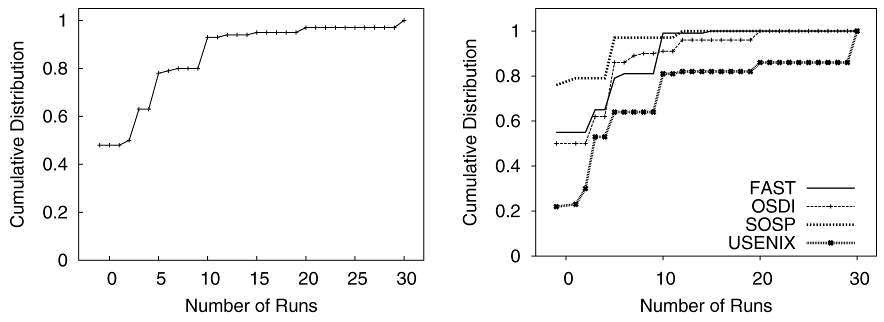
3.5 Validating Results

Other researchers may wish to benchmark one's software for two main reasons: (1) to reproduce or confirm the results, or (2) to compare their system to one's own.

First, it is considered good scientific practice to provide enough information for others to validate the results. This includes detailed hardware and software specifications about the testbeds. Although it is usually not practical to include such large amounts of information in a conference paper, these details can be published in an online appendix. Although it can be difficult for a researcher to accurately validate another's results without the exact testbed, it is still possible to see whether the results generally correlate.

Second, there may be a case where a researcher creates a system with similar properties to one's own (e.g., they are both encryption file systems), and it would be logical for the researcher to compare the two systems. However, if one's own paper showed an X% overhead over ext2 and the new file system has a Y% overhead over ext2 no claim can be made about which of the two file systems is better because the benchmarking environment is different. The researcher should benchmark both research file systems using a setup as similar as possible to that of the original benchmark. This way, both file systems are tested under the same conditions. Moreover, since they are running the benchmark in the same way as one's own paper did, no claim can be made that they chose a specific case in which their file system might perform better.

To help solve these two issues, enough information should be made available about one's testbed (both hardware and any relevant software) so that an outside researcher can validate the results. If possible, one's software should be made available to other researchers so that they can compare their system to one's own. Releasing the source is preferred, but a binary release can also be helpful if there are legal issues preventing the release of source-code. Members of the SOSP 2007 program committee attempted to improve this situation by asking authors in the submission form if they will make the source-code and raw data for their system and experiments available so that others can reproduce the results more easily. If enough authors agree to this sharing and other conferences follow suit, it may in the future become easier to compare similar systems and reproduce results. Similarly, any benchmarks written and traces collected should be made available to others.



(a) number of runs of surveyed benchmarks (b) number of runs categorized by conference

Fig. 1. (a) Cumulative distribution function (CDF) for the number of runs performed in the surveyed benchmarks; (b) CDF of the same data categorized by conference. A value of –1 was used for benchmarks where the number of runs was not specified.

4. COMPLIANCE WITH THE GUIDELINES

We now examine how well the surveyed papers followed the benchmarking practices that were discussed in Section 3. We cannot expect past papers to comply with guidelines that had not yet been published. However, before setting new guidelines for the future, it is important to see how far we are from them at the present. We also feel that a certain degree of scientific rigor in benchmarking should be a goal in one's research, even with no guidelines present.

Number of runs. Running benchmarks multiple times is important for ensuring accuracy and presenting the range of possible results. Reporting the number of runs allows the reader to determine the benchmarking rigor. We now examine the number of runs performed in each surveyed experiment. To ensure accuracy, we did not include experiments where one operation was executed many times and the per-operation latency was reported, because it was not clear whether to count the number of runs as the number of times the operation was executed, or as the number of times the entire benchmark was run. Figure 1 shows the results from the 388 benchmarks that were counted. We found that two papers [Wang et al. 2002; Wright et al. 2003b] ran their benchmarks more than once (since they included error bars or confidence intervals), but did not specify the number of runs. These are shown as two runs. The figure shows that the number of runs was not specified for the majority of benchmarks. Assuming that papers not specifying the number of runs ran their experiments once, we can break down the data by conference, as shown in Table I.

The per-conference values are presented for informational value and we feel they may of interest to the reader. However, we caution the reader against drawing conclusions based on these statistics, as benchmarking rigor alone does not determine the quality of a conference, and the number of runs alone does not determine benchmarking rigor.

Statistical dispersion. After performing a certain number of runs, it is important to inform the reader about the statistical dispersion of the results.

Table I. Percentage of Papers that Discussed Standard Deviations or Confidence Intervals (classified by conference)

	FAST	OSDI	SOSP	USENIX
Number of papers	12	21	51	23
Standard deviations	8.3%	28.6%	27.5%	69.6%
Confidence intervals	16.7%	19.1%	7.8%	8.7%
Total	25.0%	47.6%	35.3%	78.3%

Conference	Mean	Standard Deviation	Median
SOSP	2.1	2.4	1
FAST	3.6	3.6	1
OSDI	3.8	4.3	2
USENIX	4.7	6.2	3

34.6% of the surveyed papers included at least a general discussion of standard deviation, and 11.2% included confidence intervals. The percentage of papers discussing either of these aspects varied between 35.7% and 83.3% per year, but there was no upward or downward trend over time. Interestingly, we did notice significant differences between conferences, shown in Table I, but we do not suggest that this is indicative of the overall quality of any particular conference. In addition to informing the reader about the overall deviations or intervals for the paper, it is important to show statistical dispersion for each result. This can be done with error bars in graphs, by augmenting tables, or by mentioning it in the text. From all of the surveyed benchmarks, only 21.5% included this information.

Benchmark runtimes. To achieve stable results, benchmarks must run for a long enough time to reach steady state and exercise the system. This is especially important as benchmarks must scale with increasingly faster hardware. We looked at the runtimes of the 198 experiments that specified the elapsed time of the benchmark. Most benchmarks that reported only per-operation latency or throughput did not specify their runtime. For each experiment, we took the longest elapsed time of all configurations, and rounded them up to the nearest minute. For benchmarks with multiple phases, times were added to create a total time. The results are summarized in Figure 2. We can see in the figure that 28.6% of benchmarks ran for less than one min., 58.3% ran for less than five, and 70.9% ran for less than ten.

Number of benchmarks. The number of benchmarks used for performance evaluations in each paper is shown in Figure 3. We can see that 37.7% of the papers used only one or two benchmarks, which in most cases is not sufficient for a reader to fully understand the performance of a system.

System descriptions. To gain some idea of the testbed specifications published in the surveyed papers, we now present the number of parameters that were listed. It must be noted that not all parameters are equally important, and that some parameters are actually a subset of others. For example, a disk's speed is counted as one parameter, but a disk's model number is counted as one parameter as well, even though the disk's speed and several other disk

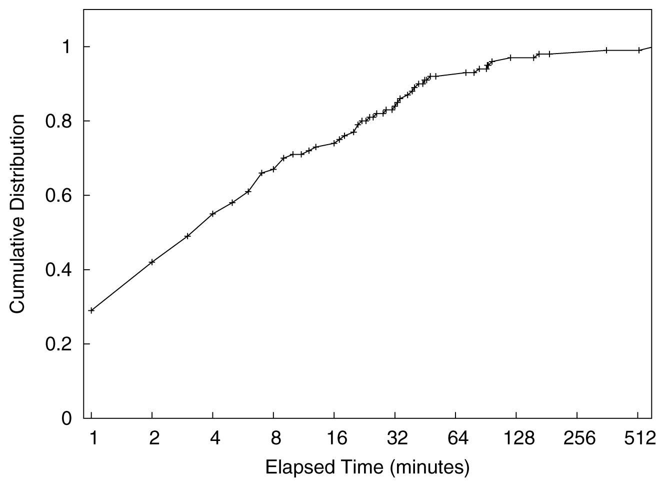


Fig. 2. CDF of the number of benchmarks run in the surveyed papers with a given elapsed time. Note the log scale on the x-axis.

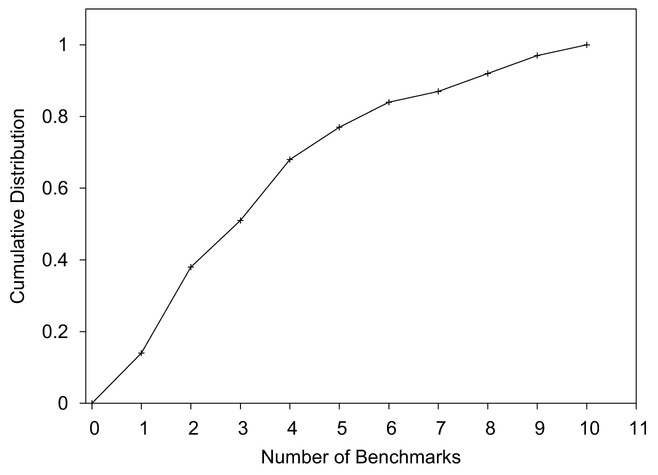


Fig. 3. CDF of the number of benchmarks run in the surveyed papers.

parameters can be found from the model specifications. Since it is not clear how to weigh each parameter, we will instead caution that these results should be used only as rough estimates. An average of 7.3 system parameters were reported per paper, with a standard deviation of 3.3. The median was 7. While this is not a small number, it is not sufficient for reproducing results. In addition, only 35.9% of the papers specified the cache state during the benchmark runs. We specify the testbed used in this article in Section 6, which we believe should be sufficient to reproduce our results.

5. RELATED WORK

A similar survey was conducted in 1997, covering more general systems papers [Small et al. 1997]. The survey included ten conference proceedings from the early to mid '90's. The main goals of that survey were to determine how reproducible and comparable the benchmarks were, as well as to discuss statistical rigor. We do not discuss statistical rigor in more detail in this article, since there is a good discussion presented there. The aforementioned paper went on to advise on how to build good benchmarks and report them with proper statistical rigor. Some results from its survey are that over 90% of file system benchmarks run were ad hoc, and two-thirds of the experiments presented a single number as a result, without any statistical information.

In 1999, Mogul presented a similar survey of general OS research papers and commented on the lack of standardization of benchmarks and metrics [Mogul 1999]. He conducted a small survey of two conference proceedings and came to the conclusion that the operating system community is in need of good standardized benchmarks. Of the eight file system papers he surveyed, no two used the same benchmark to analyze performance.

Chen and Patterson concentrated on developing an I/O benchmark that can shed light on the causes for the results, that scales well, has results that are comparable across machines, is general enough that it can be used by a wide range of applications, and is tightly specified so that everyone follows the same rules [Chen and Patterson 1993]. This benchmark does not perform metadata operations, and is designed to benchmark drivers and I/O devices. The authors go on to discuss how they made a self-scaling benchmark with five parameters: data size, average size of an I/O request, fraction of read operations (the fraction of write operations is 1 minus this value), fraction of sequential accesses (the fraction of random accesses is 1 minus this value), and the number of processes issuing I/O requests. Their benchmark keeps four of these parameters constant while varying the fifth, producing five graphs. Because self-scaling will produce different workloads on different machines, the paper discusses how to predict performance so that results can be compared, and shows reasonable ability to perform the predictions.

A paper by Tang and Seltzer from the same research group, entitled "Lies, Damned Lies, and File System Benchmarks" [Tang and Seltzer 1994], was one source for some of our observations about the Andrew (Section 7.3), LADDIS (Section 7.5), and Bonnie (Section 9.1) benchmarks, as well as serving as an inspiration for this larger study.

Tang later expanded on the ideas of that paper, and introduced a benchmark called `dtangbm` [Tang 1995]. This benchmark consists of a suite of microbenchmarks called *fsbench* and a workload characterizer. Fsbench has the four phases next described.

(1) It measures disk performance so that it can be known whether improvements are due to the disk or the file system.
(2) It estimates the size of the buffer cache, attribute cache, and name translation cache. This information is used by the next two phases to ensure proper benchmark scaling.
(3) It runs the microbenchmarks, whose results are reported. The benchmark takes various measurements within each microbenchmark, providing much information about the file system's behavior. The reported metric is in

KB/sec. The first two microbenchmarks in this phase test block allocation to a single file for sequential- and random-access patterns. The third microbenchmark tests how blocks are allocated to files that are in the same directory. The fourth microbenchmark measures the performance of common metadata operations (`create`, `delete`, `mkdir`, `rmdir`, and `stat`).

(4) It performs several tests to help file system designers to pinpoint performance problems. It isolates latencies for attribute (inode) creation, directory creation, attribute accesses, and name lookups by timing different metadata operations and performing some calculations on the results. It also uses a variety of read patterns to find cases where read-ahead harms performance. Finally, it tests how well the file system handles concurrent requests.

The second component of `dtangbm`, namely the workload characterizer, takes a trace as input and prints statistics about the operation mix, sequential versus random accesses, and the average number of open files. This information could theoretically be used in conjunction with the output from `fsbench` to estimate the file system's performance for any workload, although the authors of `dtangbm` were not able to accurately do so in that work.

Another paper from Seltzer's group [Seltzer et al. 1999] suggests that not only are the currently used benchmarks poor, but the types of benchmarks run do not provide much useful information. The current metrics do not provide a clear answer as to which system would perform better for a given workload. The common and simple workloads are not adequate, and so they discuss three approaches to application-specific benchmarking. In the first, system properties are represented in one vector and the workload properties are placed in another. Combining the two vectors can produce a relevant performance metric. The second approach involves using traces to develop profiles that can stochastically generate similar loads. The third uses a combination of both methods.

According to Ruwart, not only are the current benchmarks ill suited for testing today's systems, they will fare even worse in the future because of the systems' growing complexities (e.g., clustered, distributed, and shared file systems) [Ruwart 2001]. He discusses an approach to measuring file system performance in a large-scale, clustered supercomputer environment, while describing why current techniques are insufficient.

Finally, Ellard and Seltzer describe some problems they experienced while benchmarking a change to an NFS server [Ellard and Seltzer 2003b]. First, they describe ZCAV effects, which were previously documented only in papers that discuss file system layouts [Van Meter 1997] (not in performance evaluations). Since the inner tracks on a disk have fewer sectors than the outer tracks, the amount of data read in a single revolution can vary greatly. Most papers do not deal with this property. Aside from ZCAV effects, they also describe other factors that can affect performance, such as SCSI command queuing, disk scheduling algorithms, and differences between transport protocols (i.e., TCP and UDP).

6. BENCHMARKING METHODOLOGY

In this section, we present the testbed and benchmarking procedures that we used for conducting the experiments considered throughout the remainder of this article. Next, we describe the hardware and software configuration of the test machine as well as our benchmarking procedure.

System configuration. We conducted all our experiments on a machine with a 1.7 GHz Pentium 4 CPU, 8KB of L1 cache, and 256KB of L2 cache. The motherboard was an Intel Desktop Board D850GB with a 400 MHz system bus. The machine contained 1GB of PC800 RAM. The system disk was a 7200 RPM WD Caviar (WD200BB) with 20GB capacity. The benchmark disk was a Maxtor Atlas (Maxtor-8C018J0) 15,000 RPM, 18.4GB, Ultra320 SCSI disk. The SCSI controller was an Adaptec AIC-7892A U160.

The operating system was Fedora Core 6, with patches as of March 07, 2007. The system was running a vanilla 2.6.20 kernel and the file system was ext2 unless otherwise specified. Some relevant program versions, obtained by passing the `--version` flag on the command line, along with the Fedora Core package and version are GCC 4.1.1 (gcc.i386 4.1.1-51.fc6), GNU ld 2.17.50.0.6-2.fc6 (binutils 2.17.50.0.6-2.fc6), GNU autoconf 2.59 (autoconf.noarch 2.59-12), GNU automake 1.9.6 (automake.noarch 1.9.6-2.1), GNU make 3.81 (make 1:3.81-1.1), and GNU tar 1.15.1 (tar 2:1.15.1-24.fc6).

The kernel configuration file and full package listing are available at www.fsl.cs.sunysb.edu/project-fsbench.html.

Benchmarking procedure. We used the Autopilot v.2.0 [Wright et al. 2005] benchmarking suite to automate the benchmarking procedure. We configured Auto-pilot to run all tests at least ten times, and compute 95% confidence intervals for the mean elapsed, system, and user times using the student-*t* distribution. In each case, the half-width of the interval was less than 5% of the mean. We report the mean of each set of runs. In addition, we define "wait time" to be the time that the process was not using the CPU (mostly due to I/O).

Auto-pilot rebooted the test machine before each new sequence of runs to minimize the influence of different experiments on each other. Auto-pilot automatically disabled all unrelated system services to prevent them from influencing the results. Compilers and executables were located on the machine's system disk, so the first run of each set of tests was discarded to ensure that the cache states were consistent. We configured Auto-pilot to unmount, recreate, and then remount all tested file systems before each benchmark run. To minimize ZCAV effects, all benchmarks were run on a partition located toward the outside of the disk, and this partition was just large enough to accommodate the test data [Ellard and Seltzer 2003b]. However, the partition size was big enough to avoid the file system's space-saving mode of file system operation. In the space-saving mode, file systems optimize their operation to save disk space and thus have different performance characteristics [Van Meter 1997].

7. MACROBENCHMARKS

In this section we describe the macro-, or general purpose, benchmarks that were used in the surveyed research papers. We point out the strengths and weaknesses in each. For completeness, we also discuss several benchmarks that were not used. Macrobenchmark workloads consist of a variety of operations and aim to simulate some real-world workload. The disadvantage of macrobenchmarks is that this workload may not be representative of the workload that the reader is interested in, and it is very difficult to extrapolate from the performance of one macrobenchmark to a different workload.

Additionally, there is no agreed-upon file system benchmark that everyone can use. Some computer science fields have organizations that create benchmarks and keep them up to date (e.g., TPC in the database community). There is no such organization specifically for the file system community, although the Standard Performance Evaluation Corporation (SPEC) has one benchmark targeted for a specific network file system protocol; see Section 7.5. For storage, the Storage Performance Council [SPC 2007] has created two standardized benchmarks which we describe in Section 7.6. We have observed that many researchers use the same benchmarks, but they often neither explain the reasons for using them nor elucidate what the benchmarks show about the systems they are testing. From the 148 macrobenchmark experiments performed in the surveyed papers, 20 reported having used a benchmark because it was popular or standard, and 28 provided no reason at all. Others described what real-world workload the given benchmark was mimicking, but did not say why it was important to show these results. In total, inadequate reasoning was given for at least 32.4% of the macrobenchmark experiments performed. This leads us to believe that many researchers use the benchmarks that they are used to and that are commonly used, regardless of suitability.

In this section we describe the Postmark, various compile, Andrew, TPC, SPEC, SPC, NetNews, and other macrobenchmarks.

7.1 Postmark

Postmark [Katcher 1997; VERITAS Software 1999], created in 1997, is a single-threaded synthetic benchmark that aims at measuring file system performance over a workload composed of many short-lived, relatively small files. Such a workload is typical of electronic mail, NetNews, and Web-based commerce transactions as seen by ISPs. The workload includes a mix of data- and metadata-intensive operations. However, the benchmark only approximates file system activity; it does not perform any application processing, and so the CPU utilization is less than that of an actual application.

The benchmark begins by creating a pool of random text files with uniformly distributed sizes within a specified range. After creating the files, a sequence of "transactions" is performed (in this context a transaction is a Postmark term, and unrelated to the database concept). The number of files, number of subdirectories, file-size range, and number of transactions are all configurable. Each Postmark transaction has two parts: a file creation or deletion, operation, paired with a file read or append. The ratios of reads-to-appends and creates-to-deletes are configurable. A file creation operation creates and writes random text to a file. A file deletion operation removes a randomly chosen file from the active set. A file read operation reads a random file in its entirety and a file write operation appends a random amount of data to a randomly chosen file. It is also possible to choose whether or not to use buffered I/O.

Table II. Postmark Configuration Details

Parameter	Default Value	Number Disclosed (out of 30)
File sizes	500–10,000 bytes	21
Number of files	500	28
Number of transactions	500	25
Number of subdirectories	0	11
Read/write block size	512 bytes	7
Operation ratios	equal	16
Buffered I/O	yes	6
Postmark version	-	7

The default Postmark v1.5 configuration and the number of research papers that disclosed each piece of information (from the 30 papers that used Postmark in the papers we surveyed).

One drawback of using Postmark is that it does not scale well with the workload. Its default workload, shown in Table II, does not exercise the file system enough. This makes it no longer relevant to today's systems, and as a result researchers use their own configurations. On the machine described in Section 6, the default Postmark configuration takes less than a tenth of one sec. to run, and barely performs any I/O. One paper [Nightingale et al. 2005] used the default configuration over NFS rather than updating it for current hardware, and the benchmark completed in under seven sec. It is unlikely that any accurate results can be gathered from such short benchmark runs. In Section 12.2, we show how other Postmark configurations behave very differently from each other. Rather than having the number of transactions to be performed as a parameter, it would be more beneficial to run for a specified amount of time and report the peak transaction rate achieved. Benchmarks such as Spec SFS and AIM7 employ a similar methodology.

Having outdated default parameters creates two problems. First, there is no standard configuration, and since different workloads exercise the system differently, the results across research papers are not comparable. Second, not all research papers precisely describe the parameters used, and so results are not reproducible.

Few research papers specify all the parameters necessary for reproducing a Postmark benchmark. From the 106 research papers that we surveyed, 29 used Postmark as one of their methods for performance evaluation [Ng et al. 2002; Sarkar et al. 2003; Sivathanu et al. 2006, 2004a, 2004b; Radkov et al. 2004; Tan et al. 2005; Anderson et al. 2002; Nightingale et al. 2006, 2005; Wang et al. 2002; Seltzer et al. 2000; MacCormick et al. 2004; Abd-El-Malek et al. 2005; Thereska et al. 2004; Magoutis et al. 2002; Aranya et al. 2004; Muniswamy-Reddy et al. 2004; Zhang and Ghose 2003; Wright et al. 2003b; Strunk et al. 2000; Prabhakaran et al. 2005a; Stein et al. 2001; Soules et al. 2003; Denehy et al. 2005; Weddle et al. 2007; Schindler et al. 2002; Magoutis et al. 2003; Wang and Merchant 2007]. Table II shows how many of these papers disclosed each piece of information. Papers that use configurable benchmarks should include all the parameters to make results meaningful; only 5 did so. These 5 papers specified that any parameters not mentioned were the defaults (Table II gives them credit for specifying all parameters).

In addition to failing to specify parameters, Table II shows that only 5 out of 30 research papers mentioned the version of Postmark used. This is especially crucial with Postmark due to its major revisions that make results from different versions incomparable. The biggest changes were made in version 1.5, where the benchmark's pseudorandom number generator was overhauled. Having a generator in the program itself is a good idea, as it makes the benchmarks across various platforms more comparable. There were two key bugs with the previous pseudorandom number generator. First, it did not provide numbers that were random enough. Second, and more importantly, it did not generate sufficiently large numbers, so the files created were not as large as the parameter specified, causing results to be inaccurate at best. Having a built-in pseudorandom number generator is an example of a more general rule: Library routines should be avoided unless the goal of the benchmark is to measure the libraries because this introduces more dependencies on the machine setup (OS, architecture, and libraries).

Another lesson that Postmark teaches us is to make an effort to keep benchmarking algorithms scalable. The algorithm that Postmark uses to randomly choose files is $O(N)$ on the number of files, which does not scale well with the workload. It would be trivial to modify Postmark to fix this, but would make the results incomparable with others. While high levels of computation are not necessarily a bad quality, they should be avoided for benchmarks that are meant to be I/O bound.

An essential feature for a benchmark is accurate timing. Postmark uses the `time(2)` system call internally, which has a granularity of one sec. There are better timing functions available (e.g., `gettimeofday`) that have much finer granularity and therefore provide more meaningful and accurate results.

One of the future directions considered for Postmark is allowing different numbers of readers and writers, instead of just one process that does both. Four of the surveyed papers [Aranya et al. 2004; Anderson et al. 2004, 2002; Wang and Merchant 2007] ran concurrent Postmark processes. Seeing how multiple processes affect results is useful for benchmarking most file systems, as this reflects real-world workloads more closely. However, since Postmark is not being maintained (no updates have been made to Postmark since 2001), this will probably not be done.

One research paper introduces Filemark [Bryant et al. 2002], which is a modified version of Postmark 1.5. It differs from its predecessor in five respects. First, it adds multithreading so that it can produce a heavier and more realistic workload. Second, it uses `gettimeofday` instead of `time`, so that timing is more accurate. Third, it uses the same set of files for multiple transaction phases. This makes the runtime faster, but culminates in fewer writes, and extra care must be taken to ensure that data is not cached if this is not desired. Fourth, it allows the read-write and create-delete ratios to be specified to the nearest 1%, instead of 10% as with Postmark. Fifth, it adds an option to omit performance of the delete phase, a phase which the Filemark authors claim as having a high variation and being almost meaningless. We suggest instead that if some operation has a high variation, it should be further investigated and explained rather than discarded.

Postmark puts file systems under heavy stress when the configuration is large enough, and is a fairly good benchmark. It has good qualities such as a built-in pseudorandom number generator, but also has some deficiencies. It is important to keep in mind its positive and negative qualities when running the benchmark and analyzing results. In sum, we suggest that Postmark be improved to have a scalable workload, more accurate timing, and to allow for multithreaded workloads.

7.2 Compile Benchmarks

Of the papers we surveyed, 36 timed the compiling of some code to benchmark their projects. These papers are broken down as follows.

—Ten compiled SSH [Soules et al. 2003; Ng et al. 2002; Schindler et al. 2002; Denehy et al. 2005; Strunk et al. 2000; Prabhakaran et al. 2005a; Seltzer et al. 2000; Kroeger and Long 2001; Stein et al. 2001; Weil et al. 2006].

—Eight compiled an OS kernel [Zhang and Ghose 2003; Mazières et al. 1999; Tolia et al. 2004; Gulati et al. 2007; Radkov et al. 2004; Sivathanu et al. 2006; Papathanasiou and Scott 2004; Muniswamy-Reddy et al. 2006].

—Six papers (all from our research group) compiled Am-utils [Aranya et al. 2004; Muniswamy-Reddy et al. 2004; Zadok and Nieh 2000; Zadok et al. 2001, 1999; Wright et al. 2003b].

—Five compiled Emacs [Gulati et al. 2007; Li et al. 2004; Fu et al. 2000; Maziéres 2001; Muthitacharoen et al. 2001].

—Three compiled Apache [Peek and Flinn 2006; Nightingale et al. 2006, 2005].

—Five compiled other packages [Gulati et al. 2007; Sobti et al. 2002; Abd-El-Malek et al. 2005; Gniady et al. 2004; Lee et al. 1999].

—Two did not specify the source-code being compiled [Kroeger and Long 2001; Kim et al. 2000].

The main problem with compile benchmarks is that because they are CPU intensive, they can hide the overheads in many file systems. This issue is discussed further in Section 12.2. However, a CPU-intensive benchmark may be a reasonable choice for a file system that already has a significant CPU component (such as an encryption or compression file system). Even so, a more I/O-intensive benchmark should be run as well. Other issues relating to compile benchmarks affect the ability of readers to compare, fully understand, and reproduce benchmark results. These are presented next.

(1) Different machines may have different compiler tool chains. Specifically:
 —Different architectures produce different code.
 —Different compilers utilize different optimizations.
 —The source-code may not compile on all architectures, and older packages often cannot compile on newer systems, so the workload becomes obsolete.

(2) Different machines are configured differently (with regard to both hardware and software), so the configuration phase will not be the same on all machines, and the resulting code will be different as well.

Table III. SSH 2.1.0, Am-Utils 6.1b3, and Linux Kernel 2.4.20 Characteristics

	SSH	Am-Utils	Linux Kernel
Directories	54	25	608
Files	637	430	11,352
Lines of Code	170,239	61,513	4,490,349
Code Size (Bytes)	5,313,257	1,691,153	126,735,431
Total Size (Bytes)	9,068,544	8,441,856	174,755,840

The total size refers to the precompiled package, since the total size after compiling is system dependent.

(3) Some compilations (such as kernels) have different configuration options, resulting in a different configuration phase and different resulting code. Although default configuration files are sometimes included, using them can result in compilation errors, as we have experienced when compiling some versions of the Linux kernel.
(4) The operation mixes can change depending on which program is being compiled, and even on its version.
(5) The compile process is perpetually growing more complex and there is much variation between programs. Most require explicit configuration phases, and some require phases that resolve dependencies. The amount of time spent in each phase can also vary significantly [Zadok 2002].

To allow a benchmark to be accurately reproduced, all parameters that could affect the benchmark must be reported. This is particularly difficult with a compile benchmark. From the 33 papers that used compile benchmarks, only 1 specified the compiler and linker versions, and 1 specified compiler options. Moreover, 8 failed to specify the version of the code being compiled, and 19 failed to specify the compilation steps being measured. Although it is easy to report the source-code version, it is more difficult to specify the relevant programs and patches that were installed. For example, Emacs has dependencies on the graphical environment, which may include dozens of libraries and their associated header files. However, this specification is feasible if a package manager has been used that can provide information about all of the installed program versions. Because of the amount of information that needs to be presented, we recommend creating an online appendix with the detailed testbed setup.

There is a common belief that file systems see similar loads, independent of the software being compiled. Using OSprof [Joukov et al. 2006], we profiled the build process of three packages commonly used as compile benchmarks: (1) SSH 2.1.0; (2) Am-utils 6.1b3; and (3) the Linux 2.4.20 kernel with the default configuration. Table III shows the general characteristics of the packages. The build process of these packages consists of a configuration phase and a compilation phase. The configuration phase consists of running GNU `configure` scripts for SSH and Am-utils, and running "`make defconfig dep`" for the Linux kernel. We analyzed the configuration- and compilation phases both separately as well as together. Before the configuration- and compilation phases, we remounted the ext2 file system on which the benchmark was run, to reduce caching effects.

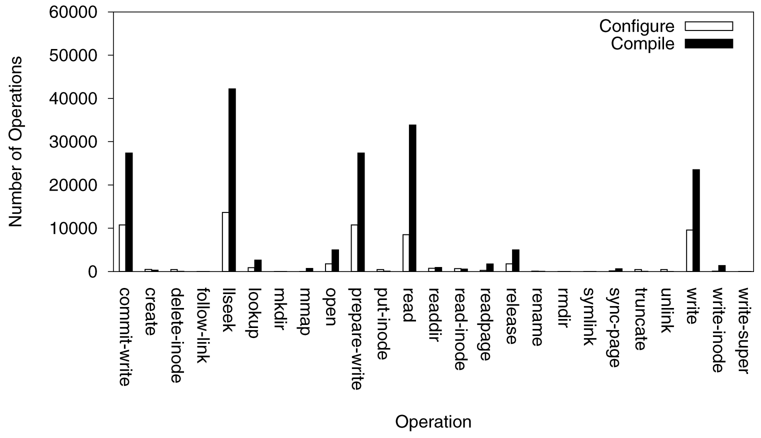
Figure 4 shows the distribution of the total number of invocations of all ext2 VFS operations used during the build processes. Note that each of the three graphs uses different scales for the number of operations (y-axis).

Figures 4(a) and 4(b) show that even though the SSH and Am-utils build process sequences, source-file structures, and total sizes appear to be similar, their operation mixes are quite different; moreover, the fact that SSH has nearly three times the lines of code of Am-utils is also not apparent from analyzing the figures. In particular, the configuration phase dominates in the case of Am-utils, whereas the compilation phase dominates the SSH build. More importantly, the read-write ratio for the Am-utils build was 0.75:1, whereas it was 1.28:1 for the SSH build. This can result in significant performance differences for read- versus write-oriented systems. Not surprisingly, the kernel-build process profile differs from both SSH and Am-utils. As can be seen in Figure 4(c), both of the kernel-build phases are strongly read biased. In addition, the kernel-build process is more intensive in file-open and file-release operations. As we can see, even seemingly similar compile benchmarks exercise the test file systems with largely different operation mixes.

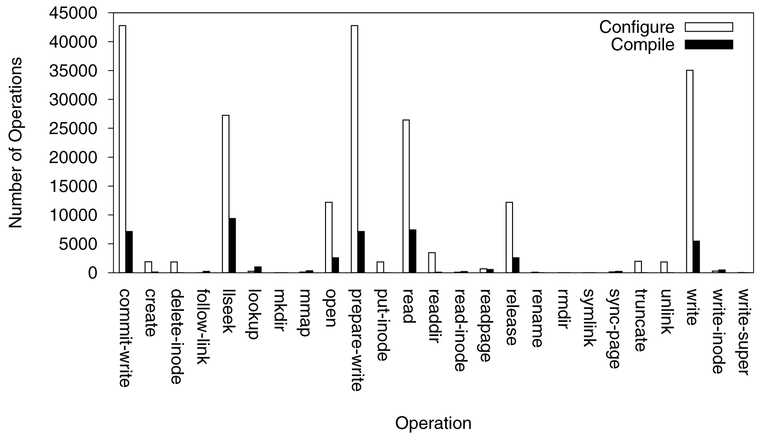
Now let us consider compilation of the same software with slightly different versions. In the paper "Opportunistic Use of Content Addressable Storage for Distributed File Systems," by Tolia et al. [2003], the authors show the commonality found between versions of the Linux 2.4 kernel source-code (from 2.4.0 to 2.4.20) and between several nightly snapshots of Mozilla binaries from March 16th, 2003 to March 25th, 2003. The commonality for both examples is measured as the percentage of identical blocks. The commonality between one version of the Linux source-code and the next ranges from approximately 72% to almost 100%, and version 2.4.20 has only about 26% in common with 2.4.0. The Mozilla binaries show us how much a normal user application can change over the course of one day: Subsequent versions had approximately 42 to 71% in common, and only about 30% of the binary remained unchanged over the course of ten days. This illustrates the point that even when performing a compile benchmark on the same program, its version can greatly affect the results.

Not only do the source-code and resulting binaries change, but the operation mixes change as well. To illustrate this point, we compiled three different, but recent, versions of SSH on our reference machine, using the same testbed and methodology described in Section 6. We used SSH because it is the most common application that was compiled in the papers we surveyed, and specifically OpenSSH because it compiles on modern systems.

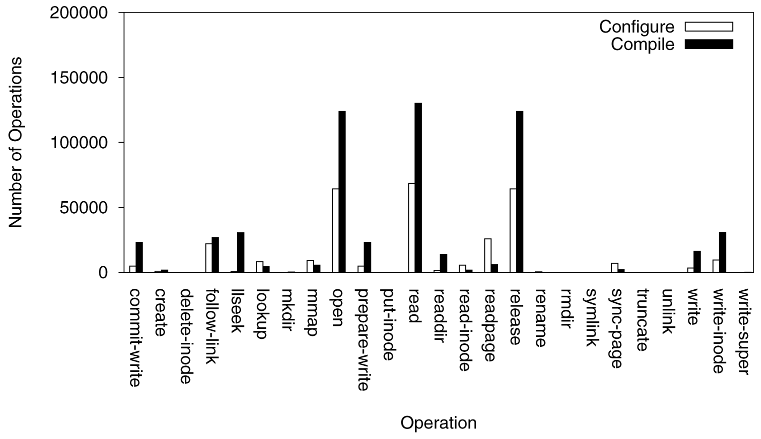
Each test consisted of unpacking, configuring, and compiling the source-code, and then deleting it. The first and last steps are less relevant to our discussion, hence we do not discuss them further. The results for the configure- and compile phases are shown in Figure 5. Although the elapsed times for the configure phase of versions 3.5 and 3.7 are indistinguishable, there is a much larger difference between versions 3.7 and 3.9 (42.3% more elapsed time, 55.6% more system time, and 25.1% more user time for the latter). There are differences between all three versions for the compile phase, with increases ranging from 6.0% to 8.4% between subsequent versions for all time components. We can see



(a) SSH 2.1.0

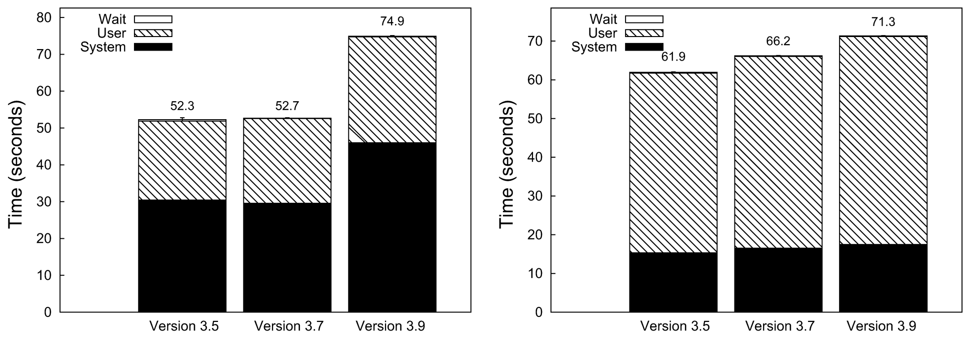


(b) Am-utils 6.1b3



(c) kernel 2.4.20

Fig. 4. Operation mixes during three compile benchmarks as seen by the ext2 file system. Note that each plot uses a different scale on the y-axis.



(a) configure phase (b) compile phase

Fig. 5. Time taken to configure and compile OpenSSH versions 3.5, 3.7, and 3.9 on ext2. Note that error bars are shown, but are small and difficult to see.

that versions of the same program that closely resemble each other are yet very different, and we can therefore infer that the difference will be greater between versions that are spread further apart, and moreso for different programs. Finally, we see how small are the effects of I/O operations on the benchmark results.

7.3 The Andrew File System Benchmark

The Andrew benchmark was created in 1988 to evaluate the performance of the Andrew file system [Howard et al. 1988]. The benchmark script operates on a directory subtree containing the source-code for a program. The operations chosen for the benchmark were intended to be representative of an average user workload [Howard et al. 1988], although this was not shown statistically accurate. Eight papers that we surveyed used this benchmark for performance analysis [Adya et al. 2002; Grönvall et al. 1999; Rhea et al. 2003; Gopal and Manber 1999; Tolia et al. 2003; Kim et al. 2002; Aguilera et al. 2003; Saito et al. 2002].

The Andrew benchmark has the five phases next presented.

(1) *MakeDir*: constructs directories in a target subtree identical to the structure of the original subtree.
(2) *Copy*: copies all files from source subtree to target subtree.
(3) *ScanDir*: performs a `stat` operation on each file of the target subtree.
(4) *ReadAll*: reads every byte of every file in the target subtree once.
(5) *Make*: compiles and links all files in the target subtree.

This benchmark has two major problems. First, the final phase of the benchmark (i.e., compilation) dominates the benchmark's runtime, thus introducing all of the drawbacks of compile benchmarks (see Section 7.2). Second, the benchmark does not scale. The default dataset will fit into the buffer cache of most systems today, so all read requests after the copy phase are satisfied without going to disk. Therefore, this does not provide an accurate picture of how the file system would behave under workloads where data is not cached. In order to resolve the issue of scalability, four of the research papers used a source program larger than that which comes with the benchmark. This, however, causes results to be incomparable between papers.

Several research papers use a modified version of the Andrew benchmark (MAB) [Ousterhout 1990] from 1990. The modified benchmark uses the same compiler so as to make the results more comparable between machines. This solves one of the issues we described when examining compile benchmarks in Section 7.2. Although using a standard compiler for all systems is a good solution, it has a drawback: The tool chain is for a machine that does not exist, and it is therefore not readily available and not maintained. This could affect usability in future machines. Seven of the research papers that we surveyed used this benchmark [Mazières et al. 1999; Muthitacharoen et al. 2002; Padioleau and Ridoux 2003; Santry et al. 1999; Nightingale and Flinn 2004; Cipar et al. 2007; Sobti et al. 2002].

One of the papers [Sobti et al. 2002] further modified the benchmark by removing the make phase and increasing the number of files and directories. Although this removes the complications associated with a compile benchmark and takes care of scalability, data can still be cached depending on the package size. Another paper [Nightingale and Flinn 2004] used Apache for the source files, and measured the time to extract the files from the archive, configure and compile the package, and remove the files. These two papers reported using a "modified Andrew benchmark," but since the term "modified" is rather ambiguous, we could not determine whether they had used the MAB compiler, or if the benchmark was called "modified" because it used a different package or had different phases.

The Andrew benchmark basically combines a compile benchmark and a microbenchmark. We suggest using separate compile benchmarks and microbenchmarks as deemed appropriate (see Sections 7.2 and 9 for extensive discussions on each, respectively).

Notable quotables. We believe some quotations from those papers that used the Andrew benchmark can provide some insight into the reasons for running it, and into the type of workload that it performs. Six of the fifteen papers that used the Andrew benchmark (or some variant) stated that it was because Andrew was popular or standard. One paper states that "primarily because it is customary to do so, we also ran a version of the Andrew benchmark" [Adya et al. 2002]. Six others gave no explicit reason for running the benchmark. The remaining three papers claimed the reason as being that its workload was representative of a user- or software developer workload.

Running a benchmark because it is popular or a standard can help readers compare results across papers. Unfortunately, this benchmark has several deficiencies. One paper states that "such Andrew benchmark results do not reflect a realistic workload" [Adya et al. 2002]. Another comments that because of the lack of I/O performed, the benchmark "will tend to understate the difference between alternatives" [Kim et al. 2002]. The authors of one paper describe that they "modified the benchmark because the 1990 benchmark does not generate much I/O activity by today standard" [Sobti et al. 2002]. Finally, one paper describes the use of the Andrew benchmark and how most read requests are satisfied from the cache.

> The Andrew Benchmark has been criticized for being old benchmark, with results that are not meaningful to modern systems. It is argued that the workload being tested is not realistic for most users. Furthermore, original Andrew [b]enchmark used a source tree which is too small to produce meaningful results on modern systems [citation removed]. However, as we stated above, the [b]enchmark's emphasis on small file performance is still relevant to modern systems. We modified the Andrew [b]enchmark to use a Linux 2.6.14 source tree [. . .]. Unfortunately, even with this larger source tree, most of the data by the benchmark can be kept in the OS's page cache. The only phase where file system performance has a significant impact is the copy phase [Cipar et al. 2007].

It seems that researchers are aware of the benchmark's drawbacks, but still use it because it has become a "standard," because its what they are accustomed to, or because it is something that other researchers are accustomed to. It is unfortunate that an inadequate benchmark has achieved this status, and we hope that a better option will soon take its place.

7.4 TPC

The Transaction Processing Performance Council (TPC) is "a no[np]rofit corporation founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry" [TPC 2005]. The organization has strict guidelines about how benchmarks are run, requires submission of the full results and configurations, and audits the results to validate them. To certify benchmark results, companies must have auditors who are accredited by the TPC board to stand by throughout the experiments. Whereas this sort of requirement is desirable in a commercial environment, it is not practical for writing academic papers. Therefore, the benchmarks are used without the accompanying strict TPC guidelines. There are four TPC benchmarks currently in use by the database community: TPC-App, TPC-C, TPC-E, and TPC-H. Here we only describe those that were used in the surveyed papers: TPC-B, TPC-C, TPC-D, TPC-H, and TPC-W.

TPC-B. This benchmark has been obsolete since 1995 because it was deemed too simplistic, but was used in one of the surveyed papers in 2005 [Prabhakaran et al. 2005a]. Another paper [Denehy et al. 2005] created a benchmark modeled after this workload. The benchmark is designed to stress-test the core of a database system by having several benchmark programs simultaneously submiting transactions of a single type as fast as possible. The metric reported is in transactions per sec.

TPC-C. Created in 1992, this benchmark and adds some complexity that was lacking in the older TPC benchmarks TPC-A and TPC-B. It is a data-intensive benchmark, portraying the activity of a wholesale supplier where a population of users executes transactions against a database. The supplier has a number of warehouses with stock, and deals with orders and payments. Five different transaction types are used, which are either executed immediately or set to be deferred. The database contains nine types of tables with various record and population sizes. The performance metric reported is transactions per min. for TPC-C (tpmC).

TPC-C was used in eight of the surveyed papers [Zhou et al. 2001; Sarkar et al. 2003; Radkov et al. 2004; Huang and Chiueh 2001; Thereska et al. 2004; Ng et al. 2002; Abd-El-Malek et al. 2005; Wachs et al. 2007]. In addition, one paper [Nightingale et al. 2006] used an implementation of TPC-C created by the Open-Source Development Lab (OSDL, which was merged into the Linux Foundation in January 2007). The OSDL has developed implementations of several TPC benchmarks [OSDL 2007]. TPC-C is being replaced by TPC-E, which is designed to be representative of current workloads and hardware, is less expensive to run because of its more practical storage requirements, and has results that are less dependent on hardware- and software configurations.

TPC-D. This benchmark was the precursor to TPC-H (explained next), and has been obsolete since 1999. This is because TPC-D was benchmarking both ad hoc queries as well as business support and reporting, and could not do both adequately at the same time. TPC-D was split into TPC-H (ad hoc queries) and TPC-R (business support and reporting).

TPC-H. The workload for this benchmark consists of executing ad hoc queries against a database and performing concurrent data modifications. Rather than being only data intensive like TPC-C, this benchmark exercises a larger portion of a database system. It uses queries and data relevant to the database community. The benchmark examines large volumes of data, executes queries with a high degree of complexity, and uses the data to give answers to critical business questions (dealing with issues such as supply and demand, profit and revenue, and customer satisfaction). The performance metric reported is called the TPC-H composite query-per-hour performance metric (QphH@Size), and reflects multiple aspects of the capability of the system to process queries, including database size, query processing power, and throughput. This benchmark was used in three of the surveyed papers [Wachs et al. 2007; Radkov et al. 2004; Gniady et al. 2004].

TPC-W. This benchmark was meant to recreate the workload seen in an Internet commerce environment. It provides little insight because it is overly complex, difficult to analyze, and does not recreate the behavior of specific applications [Schmidt et al. 2001]. TPC-W was used in one of the surveyed papers [Huang et al. 2005], but had been declared obsolete by TPC approximately six months earlier (in April 2005).

Using a benchmark that is highly regarded and monitored by a council of professionals from the database community certainly adds to its credibility. The benchmarks are kept up-to-date with new version releases, and when serious problems are found with a benchmark, it is declared obsolete. However, none of the papers that used the benchmark had its results audited, and most, if not all, did not run the benchmark according to the specifications. A drawback of using TPC benchmarks for performance analysis is that they utilize a database system, which introduces extra complexity. This makes the results less comparable between papers and makes the benchmark more difficult to set-up. Several papers opted to use traces of the workload instead (see Section 8), and one paper [Lumb et al. 2002] used a synthetic benchmark whose workload was shown to be similar to the disk traces of a Microsoft SQL server running TPC-C. Additionally, while most papers specified the database used for the experiment, barely any tuning parameters were specified, and none specified the database table layout, which can have dramatic effects on TPC benchmark performance. Although databases are known to have many tuning parameters, one paper specified only two, while others specified only one or even none. Some authors may have used the default settings since they may be less familiar with database systems than with file or storage systems, but one paper [Sarkar et al. 2003] specified that the "database settings were fine-tuned for performance" without indicating the exact settings.

7.5 SPEC

The Standard Performance Evaluation Corporation (SPEC) was founded in 1988 by a small number of workstation vendors, with the aim of creating realistic, standardized performance tests. SPEC has grown to become a successful performance standardization body with more than 60 member companies [SPEC 2005a].

SFS. The SPEC SFS benchmark [SPEC 2001; Robinson 1999] measures the performance of NFSv2 and v3 servers. It is the official benchmark for measuring NFS server throughput and response time. One of the surveyed papers [Spadavecchia and Zadok 2002] used a precursor to this benchmark called NFSSTONE [Shein et al. 1989], created in 1989 (not created by SPEC). NFSSTONE performs a series of 45,522 file system operations, mostly executing system calls, to measure how many operations per sec. an NFS server can sustain. The benchmark performs a mix of operations intended to show typical NFS access patterns [Sandberg et al. 1985]: 53% lookups, 32% reads, 7.5% readlinks (symlink traversal), 3.2% writes, 2.3% getattrs, and 1.4% creates. This benchmark performs these operations as fast as it can and then reports the average number of operations performed per sec., or NFSSTONES. The problems with this benchmark are that it only uses one client so that the server is not always saturated, it relies on the client's NFS implementation, and the file- and block sizes are not realistic.

Another benchmark called nhfsstone was developed in 1989 by Legato Systems, Inc. It was similar to NFSSTONE except that instead of the clients executing system calls to communicate with the server, they used packets created by the user-space program. This reduces the dependency on the client, but does not eliminate it because the client's behavior still depends on the kernel of the machine on which it runs.

LADDIS [Watson and Nelson 1992; Wittle and Keith 1993] was created in 1992 by a group of engineers from various companies, and was further developed when SPEC took the project over (this updated version was called SFS 1.0). LADDIS solved some of the deficiencies in earlier benchmarks by implementing the NFS protocol in user space, improving the operation mix, allowing multiple clients to generate load on the server simultaneously, providing a consistent method for running the benchmark, and porting the benchmark to several systems. Like the previous benchmarks, given a requested load (measured in operations/sec.), LADDIS generates an increasing load of operations and measures the response time until the server is saturated. This is the maximum sustained load that the server can handle under this requested load. As the requested load increases, response time diminishes. The peak throughput is reported.

LADDIS used an outdated workload, and only supported NFSv2 over UDP (no support for NFSv3 or TCP). SFS 2.0 fixed these shortcomings, but several algorithms dealing with request-rate regulation, I/O access, and the file set were found defective. SPEC SFS 3.0 fixed the latter, and updated some important features such as the time measurement.

The SFS 3.0 benchmark is run by starting the script on all clients (one will be the main client and direct the others). The number of load-generating processes, the requested load for each run, and the amount of read-ahead and write-behind are specified. For each requested load, SFS reports the average response time. The report is a graph with at least 10 requested loads on the x-axis, and their corresponding response times on the y-axis.

SPEC SFS was used by two of the surveyed papers, one of which ran in compliance with SPEC standards [Eisler et al. 2007], and one in which the matter of compliance was not clear [Anderson et al. 2000]. In addition, one paper used a variant of SFS [Patterson et al. 2002], but did not specify how it varied. One issue with SFS is that the number of systems that can be tested is limited to those that speak the NFSv2 and NFSv3 protocols. SFS cannot test changes to NFS clients, and cannot be used to compare an NFS system with a system that speaks another protocol. Whereas this benchmark is very useful for companies that sell filers, its use is limited in the research community. This limitation, combined with the fact that the benchmark is not free (it currently costs $900, or $450 for nonprofit or educational institutions), has probably impeded its widespread use in the surveyed papers.

It is of interest to note that the operation mix for the benchmark is fixed. This is good because it more greatly, standardizes the results but is also bad because the operation mix can become outdated and may not be appropriate for all settings. Some have claimed that SFS does not resemble any NFS workload they have observed, and that each NFS trace that they examined had unique characteristics, raising the question of whether one can construct a standard workload [Zhu et al. 2005a]. One can change the operations mix for SFS, so in some sense it can be used as a workload generator. However, the default, standard operations mix must be used to report any standard results that can be compared with other systems. It seems this may be the end for the SFS benchmark because NFSv4 is already being deployed, and SPEC has not stated any plans to release a new version of SFS. In addition, Spencer Shepler, one of the chairs of the NFSv4 IETF working group, has stated that SPEC SFS is "unlikely to be extended to support NFSv4," and that FileBench (see Section 10) will probably be used instead [Shepler 2005].

SDM. The SPEC SDM benchmarking suite [SPEC 2004] was made in 1991 and produces a workload that simulates a software development environment with a large number of users. It contains two subbenchmarks, 057.SDET and 061.Kenbus1, both of which feed randomly ordered scripts to the shell with commands like `make`, `cp`, `diff`, `grep`, `man`, `mkdir`, `spell`, etc. Both use a large number of concurrent processes to generate significant file system activity.

The measured metric consists of the number of scripts completed per hour. One script is generated for each "user" before the timing begins, and each contains separate subtasks executed in a random order. Each user is then given a home directory that is populated with the appropriate directory tree and files. A shell is started for each user that uses its own execution script. The timer is stopped when all scripts have completed execution.

There are two main differences between these two benchmarks. First, Kenbus simulates users typing at a rate of three characters per sec., as opposed to SDET which reads as fast as possible. Second, the command set used in SDET is a lot richer and many of the commands do a lot more work than in Kenbus.

The SDET benchmark was used in two surveyed papers [Ng et al. 2002; Seltzer et al. 2000] to measure file system performance, but not all commands executed by the benchmark exercise the file system. The benchmark is meant to measure the performance of the system as a whole, and not any particular subsystem. In addition, the benchmark description states that it exercises the `tmp` directories heavily, which means that either the benchmark needs to be changed or the file system being tested must be mounted as the system disk. However, the benchmark does give a reasonable idea of how a file system would affect everyday workloads. This benchmark is currently being updated and is being called SMT (system multitasking) [SPEC 2003], with the main goal of ensuring that all systems perform the same amount of work regardless of configuration. However, the SMT description has not been updated since 2003, so it is unknown as to whether it will be deployed.

Viewperf. The SPECviewperf benchmark [SPEC 2007] is designed to measure the performance of a graphics subsystem, and was used in one of the surveyed research papers [Gniady et al. 2004]. However, we will not delve into this benchmark's details because it is inappropriate to use it as a file system benchmark, as it exercises many parts of the OS other than the file system.

Web99. The SPECweb99 benchmark [SPEC 2005b], replaced by SPECweb2005 in 2005, is used for evaluating the performance of Web servers. One of the surveyed research papers [Nightingale et al. 2006] used it. Its workload is comprised of dynamic and static GET operations, as well as POST operations. We omit further discussion because it is a Web server benchmark, and was used by the authors of the surveyed paper to specifically measure their file system using a network-intensive workload.

7.6 SPC

The Storage Performance Council (SPC) [SPC 2007] develops benchmarks focusing on storage subsystems. Its goal is to have more vendors use industry-standard benchmarks and to publish their results in a standard way. The council consists of several major vendors in the storage industry, as well as some academic institutions. The SPC currently has two benchmarks available to its members: SPC-1 and SPC-2. Neither were used in the surveyed papers, but both are clearly noteworthy.

SPC-1. This benchmark's workload is designed to perform typical functions of business-critical applications. The workload is comprised of predominately random I/O operations, and performs both queries and update operations. This type of workload is typical of online transaction processing (OLTP) systems, database systems, or mail server applications. SPC-1 is designed to accurately measure performance and price/performance on both direct attach- or network storage subsystems. It includes three tests.

The first test phase, called "Primary Metrics," has three phases. In the first, throughput sustainability is tested for three hours at steady state. The second phase lasts for ten min. and tests the maximum attainable throughput in I/Os per sec. The third phase lasts fifty min. and maps the relationship between response time and throughput by measuring latencies at various load levels, defined as percentages of the throughput achieved in the previous phase. The third phase also determines the optimal average response time of a lightly loaded storage configuration.

The second test was designed to prove that the maximum I/O request throughput results determined in the first test are repeatable and reproducible. It does so by running similar but shorter workloads than those of the first test to collect the same metrics. In the third and final test, SPC-1 demonstrates that the system provides nonvolatile and persistent data storage. It does so by writing random data to random locations over the total capacity of the storage system for at least ten min. The writes are recorded in a log. The system is shut-down, and caches that employ battery backup are flushed or emptied. The system is then restarted, and the written data verified.

SPC-2. This benchmark is characterized by a predominantly sequential workload (in contrast to SPC-1's random one). The workload is intended to demonstrate the performance of business-critical applications that require large-scale, sequential data transfers. Such applications include large file processing (scientific computing, large-scale financial processing), large database queries (data mining, business intelligence), and on-demand video. SPC-2 includes four tests and it measures the throughput.

The first test checks data persistence, similar to the third test of SPC-1. The second test measures large file processing. It has three phases (write-only, read-write, read-only), each consisting of two run sequences, and each of these run sequences are composed of five runs (thirty runs in total). Each run consists of a certain-sized transfer with a certain number of streams. The third test is the large database query test, which has two phases (1,024KiB transfer size and 64KiB transfer size). Each phase consists of two run sequences (the first sequence consisting of four outstanding requests and the second of one outstanding request), and each sequence consists of five runs where the number of streams is varied (ten runs total). The fourth and final test is the video-on-demand delivery test, in which several streams of data are transferred.

Since the Storage Performance Council has many prominent storage vendors as members, it is likely that its benchmarks will be widely used in industry. However, their popularity in academia is yet to be seen, as the benchmarks are currently only available to SPC members, or for a cost of $500 for nonmember academic institutions. Of course, academic institutions will probably not follow all of the strict benchmark guidelines and pay the expensive result filing fees, but the benchmarks would still allow for good comparisons.

7.7 NetNews

The NetNews benchmark [Swartz 1996], created in 1996, is a shell script that performs a small-file workload comparable to that which is seen on a USENET NetNews server. It performs some setup work, and then executes the following three phases multiple times.

(1) *Unbatch*: measures the receiving and storing of new articles. Enough data is used to ensure that all caches are flushed.

(2) *Batch*: measures the sending of backlogged articles to other sites. Articles are batched on every third pass so as to reduce the runtime of the benchmark, but should be large enough such that they will not be cached when reused.

(3) *Expire*: measures the removal of "expired" articles. Since article timestamps are not relevant in a benchmark setting, the number of history entries is recorded after each unbatch phase. This information is used to delete expired articles from more than some number of previous passes. The list of articles to be deleted is written to one file, while the modified history file is written to another.

One of the surveyed papers [Seltzer et al. 2000] used this benchmark without the batch phase to analyze performance. It is a good benchmark in the sense that it is metadata intensive, and stresses the file system (given a large enough workload). However, while the data size used in the surveyed paper was considered by the authors to be large (270MB for the unbatch phase, and 250MB for the expire phase), it is much smaller than sizes seen in the real world. The paper states that "two years ago, a full news feed could exceed 2.5GB of data, or 750,000 articles per day. Anecdotal evidence suggests that a full news feed today is 15–20GB per day." This shows how a static workload from 1996 is no longer realistic just four years later. In addition to the unrealistic workload size, a USENET NetNews server workload is not very common these days, and it may be difficult to extrapolate results from this benchmark to applications that more people use.

7.8 Other Macrobenchmarks

This section describes two infrequently used macrobenchmarks that appeared in the surveyed research papers.

NetBench and Dbench. NetBench [VeriTest 2002] is a benchmark used to measure the performance of file servers. It uses a network of PCs to generate file I/O requests to a file server. According to the dbench README file [Tridgell 1999], NetBench's main drawback is that properly running it requires a lab with 60–150 PCs running Windows, connected with switched fast Ethernet and a high-end server. The dbench README file also states that since the benchmark is "very fussy," the machines should be personally monitored. Because of these factors, this benchmark is rarely used outside of the corporate world.

Dbench is an open-source program that runs on Linux machines and produces the same file system load as NetBench would on a Samba server, but without making any networking calls. It was used in one surveyed paper to measure performance [Schmuck and Haskin 2002]. The metrics reported by dbench are true throughput and the throughput expected on a Win9X machine.

Because there is no source-code or documentation available for NetBench, we were limited to analyzing the dbench source-code. The dbench program is run with one parameter: the number of clients to simulate. It begins by creating a child process for each client. There is a file that uses commands from a Windows trace of NetBench, and which each child process reads one line at-a-time. The benchmark executes the Linux equivalent of each Windows command from the trace, and the file is processed repeatedly for ten min. The main process is signaled every sec. to calculate and print the throughput up to that point, and to signal to the child processes to stop when the benchmark is over. The first two min. of the run is a warm-up period, and the statistics collected during this time are kept separate from those collected during the remainder of the run.

The main question with dbench is how closely it approximates NetBench. There are three problems we have discovered. First, there is no one-to-one correspondence between Windows and Linux operations, and the file systems natively found on each of these OSs are quite different, so it is unknown how accurate the translations. Second, NetBench has each client running on a separate machine, while Linux has the main process and all child processes running on the same machine. The added computation needed for all of the process management plus the activities of each process may affect the results. In addition, if there are many concurrent processes, the benchmark may be analyzing the performance of other subsystems (e.g., the scheduler) more than the file system. Third, dbench processes the trace file repeatedly and does not consider timings. The source of the trace file is not clear, and so it is unknown how well the operations in the trace reflect the NetBench workload. Caching of the trace file may also affect the results, since it is processed multiple times by several clients on the same machine.

8. REPLAYING TRACES

Traces are logs of operations that are collected and later replayed to generate the same workload (if done correctly). They have the same goal as macrobenchmarks have (see Section 7): to produce a workload which represents a real-world environment. However, although it may be uncertain as to whether a macrobenchmark succeeds at this, a trace will definitely recreate the workload that was traced if it is captured and played back correctly. One must ensure, however, that the captured workload is representative of the system's intended real-world environment. Of the surveyed papers, 19 used traces as part of their performance analysis [Adya et al. 2002; Sivathanu et al. 2004a, 2004b; Nightingale and Flinn 2004; Weil et al. 2006; Rowstron and Druschel 2001; Arpaci-Dusseau et al. 2003; Zhang et al. 2002; Lu et al. 2002; Lumb et al. 2003; Dimitrijevic et al. 2003; Flinn et al. 2003; Schindler et al. 2004; Tolia et al. 2004, 2003; Weddle et al. 2007; Peterson et al. 2007; Tian et al. 2007; Prabhakaran et al. 2005b].

Some use traces of machines running macrobenchmarks such as TPC-C [Dimitrijevic et al. 2003; Zhang et al. 2002], TPC-H [Schindler et al. 2004], or compile benchmarks [Weil et al. 2006]. These traces differ from the norm in that they are traces of a synthetic workload rather than a real-world environment. It is unclear why a trace of a compile benchmark was used, rather than running the compile benchmark itself. However, since the actual TPC benchmarks require a database system and have rather complicated setups, authors may opt to use traces of the TPC benchmark instead. However, it is important to replay the trace in an environment similar to where the trace was gathered. For example, one paper [Dimitrijevic et al. 2003] used a trace of a TPC run that used a file-system-based database, but the authors replayed it in an environment that bypassed the file system to access the block device directly. For more information on TPC benchmarks, see Section 7.4.

There are four problem areas with traces today, as described next in detail: the capture method, the replay method, trace realism, and trace availability.

Capture method. There is no accepted way to capture traces, and this lack can be a source of confusion. Traces can be captured at system call-, VFS-, networking-, and driver levels.

The most popular way is to capture traces at the system-call level, primarily because it is easy and the system call API is portable [Akkerman 2002; Mummert and Satyanarayanan 1994; Ousterhout et al. 1985]. One benefit of capturing at the system-call level is that this method does not differentiate between requests that are satisfied from the cache and those that are not. This allows one to test changes in caching policies. An important drawback of capturing at the system-call level is that memory-mapped operations cannot be captured. Traces captured at the VFS level contain cached and noncached requests, as well as memory-mapped requests. However, VFS tracer portability is limited even between different versions of the same OS. Existing VFS tracers are available for Linux [Aranya et al. 2004] and Windows NT [Vogels 1999; Roselli et al. 2000].

Network-level traces contain only those requests that were not satisfied from the cache. Network-level capturing is only suitable for network file systems. Network packet traces can be collected using specialized devices, or software tools like tcpdump. Specialized tools can capture and preprocess only the network-file-systems-related packets [Ellard and Seltzer 2003a; Blaze 1992]. Driver-level traces contain only noncached requests and cannot correlate the requests with the associated metadata without being provided with or inferring additional information. For example, read requests to file metadata and data-read requests cannot be easily distinguished [Ruemmler and Wilkes 1993].

The process by which the trace is captured must be explained, and should be distributed along with the trace if others will be using it. In five of the surveyed papers, the authors captured their own traces, but four of them did not specify how this was done.

When collecting file system traces for studies, many papers use tracing tools that are customized for a single study. These systems are either built in an ad hoc manner [Ousterhout et al. 1985; Roselli et al. 2000], modify standard tools [Roselli et al. 2000; Ellard et al. 2003], or are not well documented in research texts. Their emphasis is on studying the characteristics of file system operations and not on developing a systematic or reusable infrastructure for tracing. Often, the traces used excluded useful information for others conducting new studies; information excluded could concern the initial state of the machines or hardware on which the traces were collected, some file system operations and their arguments, pathnames, and more.

Replay method. Replaying a file system trace correctly is not as easy as it may appear. Before one can start replaying, the trace itself may need to be modified. Any missing operations have to be guessed so that operations that originally succeeded do not fail (and those that failed should not succeed) [Zhu et al. 2005a]. For example, files must be created before they are accessed. In addition, the trace may be scaled spatially or temporally [Zhu et al. 2005a]. For parallel applications, finding internode dependencies and inter-I/O compute times in the trace improves replay correctness [Mesnier et al. 2007]. Once the trace is ready, the target file system must be prepared. Files assumed to exist in the trace must exist on the file system with at least the size of the largest offset accessed in it. This will ensure that the trace can be replayed, but the resulting file system will have no fragmentation and will only include those files that were accessed in the trace, which is not realistic. The solution here is to age the file system. Of course, the aging method should be described because the replay will then differ from the original run, as well as from replays that aged the file system differently. There are fewer issues with preparing block-level traces for replay on traditional storage devices, since block accesses generally do not have associated context. In this case, missing operations cannot be known and aging the storage system will not affect the behavior of the benchmark, since the blocks specified by the trace will be accessed regardless of the storage system's previous state.

It is natural to replay traces at the level at which the traces were captured. However, replaying file system traces at the system-call level makes it impossible to replay high I/O-rate traces with the same speed as they were captured on the same hardware. This is because replaying adds overheads associated with system calls (context switches, verifying arguments, copies between user space and kernel space [Anderson et al. 2004]). VFS-level replaying requires kernel-mode development, but can use the time normally spent on executing system calls to prefetch and schedule future events [Joukov et al. 2005].

Network-level replaying is popular because it can be done entirely from the user level. Unfortunately, it is only applicable to network file systems. Driver-level replaying allows one to control the physical data position on the disk, and is often done from user level.

Replay speed is an important consideration, and is subject to some debate. Some believe that the trace should be replayed with the original timings, although none of the surveyed papers specified having done so. There are replaying tools such as Buttress [Anderson et al. 2004] that have been shown to follow timings accurately. However, with the advent of faster machines, it would be unreasonable to replay an older trace with the same timings. On the other hand, if the source of the trace was a faster machine, it may not be possible to use the same timings.

There is another school of thought that believes that the trace should be played as fast as possible, ignoring the timings. Five of the surveyed papers did so [Flinn et al. 2003; Tolia et al. 2004; Peterson et al. 2007; Nightingale and Flinn 2004; Prabhakaran et al. 2005b]. Any trace replay speed will measure something slightly different than the original system's behavior when the trace was being captured. However, replaying a trace as fast as possible changes the behavior more than other speeds do, due to factors such as caching, read-ahead, and interactions with page write-back and other asynchronous events in the OS. It assumes an I/O bottleneck, and ignores operation dependencies.

A compromise between using the original timings and ignoring them is to play back the trace with some speedup factor. Three of the surveyed papers [Zhang et al. 2002; Sivathanu et al. 2004b; Weddle et al. 2007] replayed with both the original timings as well as at increased speeds. By doing so they were able to observe the effects of increasing the pressure on the system. Although this is better than replaying at only one speed, it is not clear what scaling factors to choose.

We believe that currently the best option is to replay the trace as fast as possible and report the average operations per sec. However, it is crucial to respect dependencies in file system traces, and not simply run one operation after the other. For example, TBBT [Zhu et al. 2005a] has one replay mode called "conservative order" which sends out a request only after all previous operations have completed, and another called "FS dependency order" which applies some file system orderings to the operations (e.g., it will not write to a file before receiving a reply that the file was created).

An ideal way of recreating a traced workload would be to first normalize the think times using both the hardware- and software (e.g., OS, libraries, relevant programs) specifications of the system that produced the trace, and then to calibrate it using specifications of the machine being used to replay the trace. How to do this accurately is still an open question, and the best we can do right now is take the results with a degree of skepticism.

However replaying is done, the method should be clearly stated along with the results. Of the 19 surveyed papers that utilized tracing for performance analysis, we found that 15 did not specify the tool used to replay and 11 did not specify the speed. Not specifying the replay tool hinders the reader from judging the accuracy of the replay. For those that did not specify the speed, one can guess that traces were replayed as fast as possible since this is common and easy, but it is still possible that other timings were used.

Trace availability. Researchers may collect traces themselves, request them directly from other researchers, or obtain them from some third party that makes traces publicly available. Reusing traces, where appropriate, can encourage comparisons between papers and allow results to be reproduced. However, traces can become unavailable for several reasons, some of which are discussed here. One surveyed paper [Rowstron and Druschel 2001] used traces that are available via FTP, but the company that captures and hosts the traces states that they remove traces after seven days. In some cases, those who captured the traces have moved on and are unavailable. Additionally, since trace files are usually very large (traces from HP Labs, which are commonly used, can be up to 9GB), researchers may not save them for future use. Some traces are even larger (approximately 1TB, compressed 20–30:1), so even if the authors still have the trace, they may insist on transferring it by shipping a physical disk. To resolve these types of issues, traces should be stored in centralized, authoritative repositories of traces for all to use. In 2003, the Storage Network Industry Association (SNIA) created a technical working group called IOTTA (I/O traces, tools and analysis) to attack this problem. They have established a worldwide trace repository, with several traces in compatible formats and with all of the necessary tools [SNIA 2007]. There are also two smaller repositories hosted by universities [LASS 2006; PEL 2001].

Privacy and anonymization is a concern when collecting and distributing traces, but it should be done while not harming the usability of the traces. For example, one could encrypt sensitive fields, each with a different encryption key. Different mappings for each field remove the possibility of correlation between related fields. For example, UID = 0 and GID = 0 usually occur together in traces, but this cannot be easily inferred from anonymized traces in which the two fields have been encrypted using different keys. Keys could be given out in private to decrypt certain fields if desired [Aranya et al. 2004]. Although this does not hide all the important information, such as the number of users on the system, it should provide enough privacy for most scenarios.

Realism. Whether a trace accurately portrays the intended real-world environment is an important issue to consider. One aspect of this problem is that of traces becoming stale. For traces whose age we could determine, the average age was 5.7 years, with some as old as 11 years. Studies have shown that characteristics such as file sizes, access patterns, file system size, file types, and directory size distribution have changed over the years [Roselli et al. 2000; Agrawal et al. 2007]. While it is acceptable to use older traces for comparisons between papers, we recommend that researchers include results that use new traces as well. We hope that new traces will be continuously collected and made available to the public via trace repositories.

Additionally, the trace should ideally be of many users executing similar workloads (preferably unaware that the trace is being collected so that their behavior is not biased). For example, one of the surveyed papers [Adya et al.

2002] used a one-hour-long trace of one developer as the backbone of their evaluation. Such a small sample may not accurately represent a whole population.

9. MICROBENCHMARKS

In this section we describe the microbenchmarks used in the surveyed research papers, and reflect on their positive qualities, drawbacks, and how appropriate they were in the context of the papers in which they appeared.

In contrast to the macrobenchmarks described in Section 7, microbenchmark workloads usually consist of a small number of types of operations and serve to highlight some specific aspect of the file system.

We discuss Bonnie and Bonnie++, the Sprite benchmarks, ad hoc microbenchmarks, as well as using system utilities to create workloads.

9.1 Bonnie and Bonnie++

Bonnie, developed in 1990, performs a series of tests on a single file which is 100KB by default [Bray 1996]. For each test, Bonnie reports the number of bytes processed per elapsed sec. the number of bytes processed per CPU sec. and the percentage of CPU usage (user and system). The tests are as follows.

—*Sequential Output.* The file is created one character at-a-time, and then recreated one 8KB chunk at-a-time. Each chunk is then read, dirtied, and rewritten.

—*Sequential Input.* The file is read once one character at-a-time, and then again one chunk at-a-time.

—*Random Seeks.* A number of processes seek to random locations in the file and read a chunk of data. The chunk is modified and rewritten 10% of the time. The documentation states that the default number of processes is four, but we have checked the source-code and only three are created (this shows the benefit of having open-source benchmarks). The default number of seeks for each process is 4,000 and is one of several hard-coded values.

Even though this is a fairly well-known benchmark [Bryant et al. 2001], of all the papers that we surveyed, only one [Zadok et al. 2001] used it. Care must be taken to ensure that the working file size is larger than the amount of memory on the system so that not all read requests are satisfied from the page cache. The Bonnie documentation recommends using a file size that is at least 4 times bigger than the amount of available memory. However, the biggest file size that was used in the surveyed research paper was equal to the amount of memory on the machine. The small number of papers using Bonnie may be due to the three drawbacks it has, explained next.

First, unlike Postmark (see Section 7.1), Bonnie does not use a single pseudorandom number generator for all OSs. This injects some variance between benchmarks run on different OSs, and results may not be comparable.

Second, the options are not parameterized [Bryant et al. 2001]. Of all of the values mentioned before, only file size is configurable from the command line; the rest of the values are hard-coded in the program. In addition, Bonnie does not allow the workload to be fully customized. A mix of sequential and random access is not possible, and the number of writes can never exceed the number of reads (because a write is done only after a chunk is read and modified).

Third, reading and writing one character at-a-time tests the library call throughput more than the file system because the function that Bonnie calls (`getc`) uses buffering.

Bonnie++ [Coker 2001] was created in 2000 and used in one of the surveyed papers [Peterson et al. 2005]. It differs from Bonnie in three ways. First, it is written in C++ rather than C. Second, it uses multiple files to allow accessing datasets larger than 2GB. Third, it adds several new tests that benchmark the performance of `create`, `stat`, and `unlink`. Although it adds some useful features to Bonnie, Bonnie++ still suffers from the same three drawbacks of Bonnie.

9.2 Sprite LFS

Two microbenchmarks from the 1992 Sprite LFS file system [Rosenblum 1992] are sometimes used in research papers for performance analysis: the large-file benchmark and the small-file benchmark.

Sprite LFS Large-File Benchmark. Three papers [Mazières et al. 1999; Wang et al. 2002, 1999] included this benchmark in their performance evaluation. The benchmark has five phases.

(1) creates a 100MB file using sequential writes;
(2) reads the file sequentially;
(3) writes 100MB randomly to the existing file;
(4) reads 100MB randomly from the file; and
(5) reads the file sequentially.

The most apparent fault with this benchmark is that it uses a fixed-size file and therefore does not scale. It also uses the `random` library routine rather than having a built-in pseudorandom number generator (as Postmark does; see Section 7.1), which may make results incomparable across machines with different implementations.

Another of its faults is that the caches are not cleaned between each phase, and so some number of operations in a given phase may be serviced from the cache (the amount would actually depend on the pseudorandom number generator for phases 3 and 4, further emphasizing the need for a common generator). However, two of the papers did specify that caches were cleaned after each write phase. It should be noted that for the random-read phase, the benchmark ends up reading the entire file, and so the latency of this phase depends on the file system's read-ahead algorithm (and the pseudorandom number generator).

One of the good points of the benchmark is that each stage is timed separately with a high level of accuracy, and only relevant portions of the code are timed. For example, when performing random writes, it first generates the random order (though it uses a poorly designed algorithm which is $O(N^2)$ in the worst case), and then starts timing the writes.

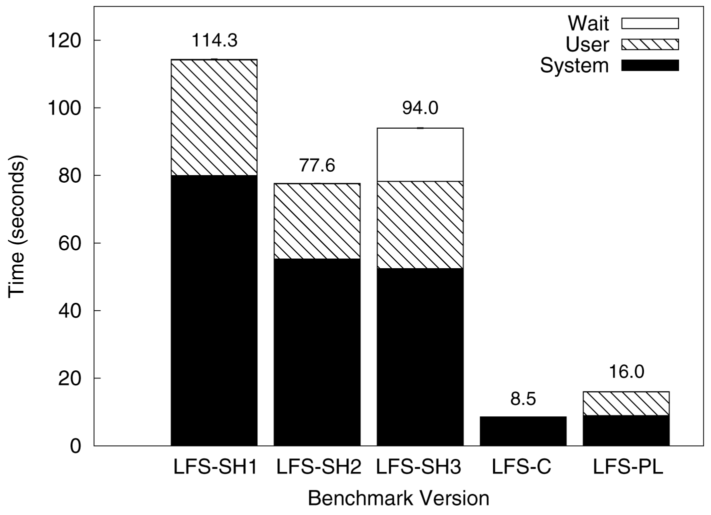


Fig. 6. Time taken to execute various versions of the Sprite LFS small-file benchmark. Note that error bars are shown, but are small and difficult to see.

Sprite LFS Small-File Benchmark. This benchmark was used in six papers [Wang et al. 2002, 1999; Denehy et al. 2005; Mazières et al. 1999; Kaminsky et al. 2003; Li et al. 2004]. It has three phases:

(1) creating 10,000 1KB files by creating and opening a file, writing 1KB of data, and closing the file;
(2) reading the files; and
(3) deleting the files.

Some papers varied the number of files and their sizes, and two specified that the caches were flushed after the write phase. We could not obtain the source-code for this benchmark, so it seems that each author may rewrite it because it is so simple. This would make it difficult to compare results across papers because the source-code may be different. To show this, we have developed the following five versions of the code.

—*LFS-SH1.* This is a `bash` script that creates the files by coping data from `/dev/zero` using the `dd` program (with a block size of 1 byte and count of 1,024), reads the files using `cat`, and deletes them with `rm`.
—*LFS-SH2.* This version is similar to LFS-SH1, but `dd` uses a 1,024-byte block size, and a count of 1.
—*LFS-SH3.* This code is the same as LFS-SH1, but uses `cp` instead of `dd` to create the files.
—*LFS-C.* This is a C implementation of the code.
—*LFS-PL.* This is a Perl code implementation.

The source-code for all five versions is available at www.fsl.cs.sunysb.edu/project-fsbench.html. The results, shown in Figure 6, clearly demonstrate that different implementations yield significantly different results. The three `bash` script versions are much slower than the others because every operation in the benchmark forks a new process. In addition, all of the implementations except for LFS-SH3 have insignificant wait-time components, showing that file system activity is minimal.

9.3 Ad Hoc MicroBenchmarks

Until now, we have been discussing widely available benchmarks in isolation. In contrast, ad hoc benchmarks are written by the authors for in-house use. In this section, we describe ad hoc microbenchmarks in the context of the papers in which they appear, since the benchmarks alone are usually not very interesting. 62 of the 106 surveyed research papers used ad hoc microbenchmarks for at least one of their experiments (191 total ad hoc microbenchmarks).

These benchmarks all have a general drawback. Because they are not made available to other researchers, they are not reproducible. Even if they are described in detail (which is usually not the case), another implementation will certainly differ (see Section 9.2 for experimental evidence). In addition, since these benchmarks are not widely used, they are not tested as much as widely available benchmarks, and therefore are more prone to bugs. One good aspect we have noticed about these benchmarks is that usually some reasoning behind the benchmark is described.

Because microbenchmarks test the performance of a file system under very specific circumstances, they should not be used alone to describe general performance characteristics. The only exception to this rule is if a minor change is made in some code and there is a clear explanation as to why no other execution paths are affected.

We have identified three reasonable ways of using microbenchmarks. The first acceptable way of using ad hoc microbenchmarks is for the purpose of better understanding the results of other benchmarks. For example, one paper [Ng et al. 2002] measured the throughput of reads and writes for sequential-, random-, and identical-block access for this purpose. Because the results are not meant to be compared across papers, the reproducibility is no longer much of an issue. Sequential access may show the best case because of its short disk-head seeks and predictable nature, a domain where read-ahead and prefetching shine. The random-read microbenchmark is generally used to measure read throughput in a scenario where there is no observable access pattern and disk-head seeks are common. This type of behavior has been observed in database workloads. The random aspect of the benchmark inherently inhibits its reproducibility across machines, as discussed in Section 7.1. Since the aforesaid paper involved network storage, repeatedly reading from the same block gives a time for issuing a request over the network that results in a cache hit. The reason for writing to the same block, however, was not explained. Nevertheless, using ad hoc microbenchmarks to explain other results is a good technique. Another paper [Fu et al. 2000] used the read phases of LFS benchmarks to examine performance bottlenecks.

A second method for using these benchmarks is to employ several ad hoc microbenchmarks to analyze the performance of a variety of operations, as eight papers did. This can provide a sense of how the system would perform compared to some baseline for commonly used operations, and may allow readers to estimate the overheads for other workloads.

The third way to acceptably use the microbenchmark is in order to isolate a specific aspect of the system. For example, a tracing file system used an ad hoc microbenchmark to exercise the file system by producing large traces that general-purpose benchmarks such as Postmark could not produce, thereby showing worst-case performance [Aranya et al. 2004]. Another used a simple sequential-read benchmark to illustrate different RPC behaviors [Magoutis et al. 2003]. Others used ad hoc microbenchmarks to show how the system behaves under specific conditions. Most of these papers focused on the read-, write-, and stat operations, varying the access patterns, number of threads, and number of files. However, most did not use the microbenchmarks to show worst-case behavior.

In addition, ad hoc microbenchmarks can be used in the initial phases of benchmarking to explore the behavior of a system. This can provide useful data about code that requires optimization, or to make decisions about what additional benchmarks would most effectively show the system's behavior.

9.4 System Utilities

Some papers use standard utilities to create workloads instead of creating workloads from scratch, as discussed in Section 9.3. Some examples of benchmarks of this type that were used in the surveyed papers are next provided.

- `wc` [Van Meter and Gao 2000] and `grep` [Van Meter and Gao 2000; Fraser and Chang 2003] sequentially read from one or more files.
- `cp` [Zadok et al. 2001; Padioleau and Ridoux 2003; Schindler et al. 2002; Santry et al. 1999; Muniswamy-Reddy et al. 2004] sequentially reads from one file while copying to another.
- `diff` [Schindler et al. 2002] sequentially reads two files and compares them.
- `tar` [Lee et al. 1999; Anderson et al. 2000] and `gzip` [DeBergalis et al. 2003] sequentially read from a file and append to a set of files while consuming CPU.

Using these utilities is slightly better than creating ad hoc benchmarks because the former are widely available and there is no misunderstanding about what the benchmark does. However, none of the papers specified what version the authors were using, which could lead to some (possibly minor) changes in workloads; for example, different versions of `grep` use different I/O strategies. However, researchers can easily specify their tool version and thus eliminate all ambiguity. Nonetheless, an important flaw in using these utilities is that the benchmarks do not scale, and depend on input files which are not standardized.

10. CONFIGURABLE WORKLOAD GENERATORS

Configurable workload generators generally have lower flexibility when compared to creating custom benchmarks, but they require less setup time and are usually more reproducible. In addition, since they are more widely used and established than ad hoc benchmarks, it is likely that they contain fewer bugs. We discuss some of the more popular generators here.

Iometer. This workload generator and measurement tool was developed by Intel in 1998 and originally developed for Windows [OSDL 2004]. Iometer was given to the Open-Source Development Lab in 2001, which open-sourced and ported it to other OSs. The authors claim that it can be configured to emulate the disk or network I/O load of any program or benchmark, and that it can be used to generate entirely synthetic I/O loads. It can generate and measure loads on single- or multiple (networked) systems. Iometer can be used for measurement and characterization of disk- and network controller performance, bus latency and bandwidth, network throughput to attached drives, shared-bus performance, and hard drive- and network performance.

The parameters for configuring tests include the following: the runtime, the amount of time to run the benchmark before collecting statistics (useful for ensuring that the system is in a "steady state"), the number of threads; number of targets (i.e., disks or network interfaces), number of outstanding I/O operations, and the workload to run.

The parameters for a thread's workload include: the percent of transfers of a given size, the ratio of reads-to-writes, the ratio of random-to-sequential accesses, number of transfers in a burst, time to wait between bursts, the alignment of each I/O on the disk, and the size of the reply (if any) to each I/O request. The test also includes a large selection of metrics to use when displaying results, and can save and load configuration files.

Iometer has four qualities not found in many other benchmarks. First, it scales well, since the user gives as input the amount of time the test should run, rather than the amount of work to be performed. Second, allowing the system to reach steady state is a good practice, although it may be more useful to find this point by statistical methods rather than by trusting the user to input a correct time. Third, it allows for easily distributing and publicizing the configuration files by saving them so that benchmarks can be run with exactly the same workloads. Although researchers can publicize parameters for other benchmarks, there is no standard format so some parameters are bound to be left unreported. Fourth, having a suite that runs multiple tests with varying parameters saves time and reduces errors. However, there are tools such as Auto-pilot [Wright et al. 2005] that can automate benchmarks with greater control (e.g., the machine can reboot automatically between runs, run helper scripts, etc.).

A drawback of Iometer is that it does not leave enough room for customization. Although it can recreate most commonly used workloads, hard-coding the possibilities for workload specification- and performance metrics reduces its flexibility. For example, the percentage of random reads is not sufficient to describe all read patterns. One read pattern suggested for testing read-ahead is reading blocks from a file in the following patterns: 1, 51, 101, . . . ; 1, 2, 51, 52, 101, 102, . . . ; 1, 2, 3, 51, 52, 53, 101, 102, 103, . . . [Tang 1995]. Such patterns cannot be recreated with Iometer.

Surprisingly, even though Iometer has many useful features, only two papers used it [Sarkar et al. 2003; Yu et al. 2000]. This may be because Iometer was unable to generate the desired workload, as described earlier. However, most workloads are fairly straightforward, so this is less of a factor. More likely, researchers simply may not know about or be familiar with it. This may be why researchers prefer to write their own microbenchmarks rather than using a workload generator. Furthermore, it does not seem that the ability to save configuration files improved the reporting of workloads in the research papers that used Iometer (neither paper fully described their microbenchmarks).

Buttress. The goal of Buttress is to issue I/O requests with a high accuracy, even when high throughputs are requested [Anderson et al. 2004]. This is important for Buttress's trace replay capability, as well for obtaining accurate inter-I/O times for its workload generation capability. Accurate I/O issuing is not present in most benchmarks, and the authors show how important this is to have. The Buttress toolkit issues read- and write requests close to their intended issue time, can achieve close to the maximum possible throughput, and can replay I/O traces as well as generate synthetic I/O patterns. Several interesting techniques were employed to ensure these properties. In addition, the toolkit is flexible (unlike Iometer) because users can specify their own workloads using a simple event-based programming interface. However, this also makes it more difficult to reproduce benchmarks from other papers (it is easier to specify simple parameters, as with Iometer). Two of the surveyed research papers, both from HP Labs, have used this toolkit [Lumb et al. 2003; Lu et al. 2002]. Unfortunately, Buttress is only available by special request from HP.

FileBench. This workload generator from Sun Microsystems [McDougall and Mauro 2005] is configured using a scripting language. FileBench includes scripts that can generate application-emulating or microbenchmark workloads, and users may write their own scripts for custom benchmarks. The application workloads it currently emulates consist of an NFS mail server (similar to Postmark; see Section 7.1), a file server (similar to SPEC SFS; see Section 7.5), a database server, a Web server, and a Web proxy.

FileBench also generates several microbenchmark workloads, some of which are similar to Bonnie (see Section 9.1) or the copy phase of the Andrew benchmark (see Section 7.3). In addition to the workloads that come with FileBench, it has several useful features: (1) workload scripts that can easily be reused and published, (2) the ability to choose between multiple threads or processes, (3) microsec. accurate latency- and cycle counts per system call, (4) thread synchronization, (5) warm-up and cool-down phases to measure steady-state activity, (6) configurable directory structures, and (7) database emulation features (e.g., semaphores, synchronous I/O, etc.).

Only one of the surveyed papers [Gulati et al. 2007] used FileBench, possibly because Filebench was made publicly available only in 2005 on Solaris, though a Linux port was created soon after. However, it is highly configurable, and it is possible that researchers will be able to use it for running many of their benchmarks.

Fstress. This workload generator has similar parameters to other generators [Andrerson 2002]. One can specify the following: the distributions for file, directory, and symlink counts, the maximum directory-tree depth, popularity in accesses for newly created objects, and file sizes, operation mixes, I/O sizes, and load levels. Like SPEC SFS (see Section 7.5), it only runs over NFSv3, and constructs packets directly rather than relying on a client implementation. However, NFSv3 is currently being replaced by NFSv4, so supporting this protocol would be necessary to ensure relevance. Like Iometer, it has limited workload configuration parameters. Another drawback is that requests are sent at a steady rate, so bursty I/O patterns cannot be simulated. Fstress was not used in any of the surveyed papers.

11. BENCHMARKING AUTOMATION

Proper benchmarking is an iterative process. In our experience, there are four primary reasons for this. First, when running a benchmark against a given configuration, one must run each test a sufficient number of times to gain confidence that the results are accurate. Second, most software does not exist in a vacuum; there is at least one other related system or a system that serves as a baseline for comparison. In addition to one's own system, one must benchmark the other systems and compare one's own system performance to those. Third, benchmarks often expose bugs or inefficiencies in one's code, which requires changes. After fixing these bugs (or simply adding new features), one must rerun the benchmarks. Fourth, after doing a fair number of benchmarks, you inevitably run into unexpected, anomalous, or just interesting results. To explain these results, one often needs to change configuration parameters or measure additional quantities, necessitating additional iterations of one's benchmark. Therefore, it is natural to automate the benchmarking process from start to finish.

Auto-pilot [Wright et al. 2005] is a suite of tools that we developed for producing accurate and informative benchmark results. We have used Auto-pilot for over five years on dozens of projects. As every project is slightly different, we continuously enhanced Auto-pilot and increased its flexibility for each. The result is a stable and mature package that saves days and weeks of repetitive labor on each project. Auto-pilot consists of four major components: a tool to execute a set of benchmarks described by a simple configuration language, a collection of sample shell scripts for file system benchmarking, a data extraction and analysis tool, and a graphing tool. The analysis tool can perform all of the statistical tests that we described in Section 3.

12. EXPERIMENTAL EVALUATIONS

In this section we describe the methods we used to benchmark file systems to show some of their qualities. Our goal is to demonstrate some of the common pitfalls of file system benchmarks. We describe the file system that we used for the benchmarks and show some of the faults that exist in commonly used benchmarks.

12.1 Slowfs

To reveal some characteristics of various benchmarks, we have modified the ext2 file system to slow down certain operations. In the remainder of this article, we call this modified ext2 file system *Slowfs*. Rather than calling the normal function for an operation, we call a new function which does the following.

(1) *start* = getcc() [get current time in CPU cycles];
(2) calls the original function for the operation;
(3) *now* = getcc();
(4) $goal = now + ((now - start) * 2^N) - (now - start)$; and
(5) while (getcc() $\leq$ goal) { schedule() }.

The net effect of this is a slowdown of a factor of 2^N for the operation. The operations to slowdown and N are given as mount-time parameters. For this article we slowed down the following operations.

—*Read.* Reads data from the disk.
—*Prepare Write and Commit Write.* These operations are used by the file system to write data to disk for the write system call. We refer to these operations collectively as WRITE for the remainder of the article.
—*Lookup.* Takes a directory and a file name, and returns an in-memory inode.

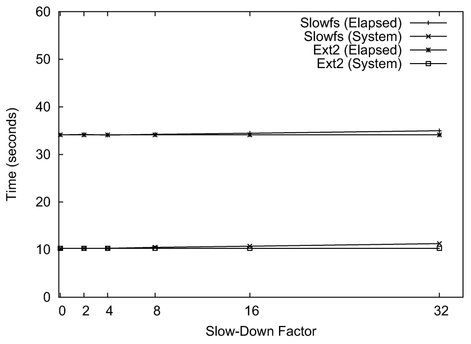
If no operation was slowed down, we call it EXT2. If all of the aforesaid operations were slowed down we call it ALL. We experimented with the preceding three functions because they are among the most common found in benchmarks. Note that this type of slowdown exercises the CPU and not I/O, and that a slowdown of a certain factor is as seen inside the file system, not by the user (the amount of overhead as seen by the user varies with each benchmark). For example, heavy use of the CPU can be found in file systems that perform encryption, compression, or checksumming for integrity or duplicate elimination.

The source-code for Slowfs is available at www.fsl.cs.sunysb.edu/project-fsbench.html.

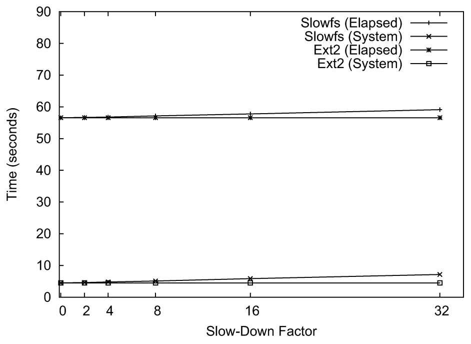
12.2 Hiding Overheads

In this section we use Slowfs to prove some of the claims we have made in this article.

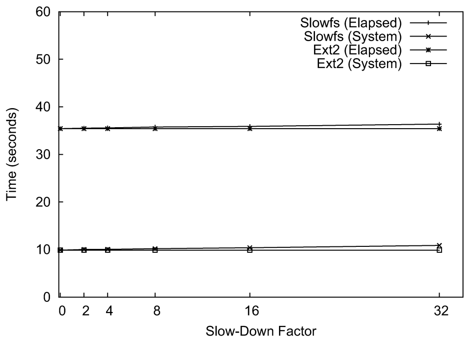
Compile benchmarks. In our first experiment, we compared Slowfs and ext2 for configuring and compiling OpenSSH versions 3.5, 3.7, and 3.9. We used Slowfs with the read operation slowed down by several factors. The results are shown in Figure 7. Only the results for versions 3.5 and 3.7 are shown because they showed the highest overheads. We chose to slow down the read operation because, as shown in Section 7.2, it is the most time-consuming operation for this benchmark. Because a compile benchmark is CPU intensive, such extraordinary overheads as a factor of 32 on read can go unnoticed (the factor of 32 comes from setting N to 5, as described in Section 12.1). For all of these graphs, the half-widths were less than 1.5% of the mean, and the *CPU%* was always



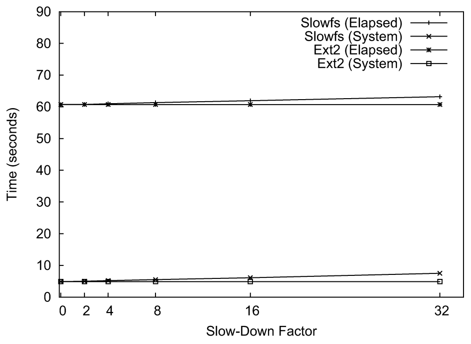
(a) configure phase for version 3.5



(b) compile phase for version 3.5



(c) configure phase for version 3.7



(d) compile phase for version 3.7

Fig. 7. Time taken to configure and compile OpenSSH versions 3.5 and 3.7 on ext2 and on Slowfs with the read operation slowed down by several factors. Note the different scales for the y-axes. The half-widths were always less than 1.5% of the mean.

more than 99.2%, where $CPU\% = \frac{time_{user} + time_{system}}{time_{elapsed}} \times 100$. In the following discussion, we do not include user or I/O times because they were always either statistically indistinguishable or very close (these two values were not affected by the Slowfs modifications).

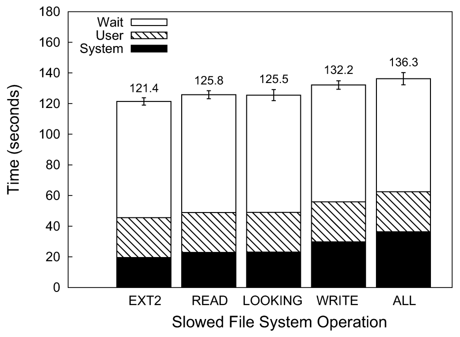
For the configure phase, the highest overhead was 2.7% for elapsed time, and 10.1% for system time (both for version 3.7). For the compile phase, the highest overhead was 4.5% for elapsed time and 59.2% for system time (both for version 3.5). Although 59.2% is a noticeable overhead, this can be hidden by only reporting the elapsed time overhead.

We also conducted the same compile benchmarks with slowing down each of the operations listed in Section 12.1 by only a factor of five. We slowed them down both separately as well as together. There was *no* statistical difference between ext2 and any of these slowed down configurations.

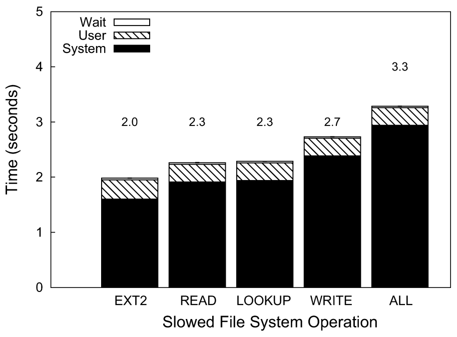
These results clearly show that even with extraordinary delays in critical file system operations, compile benchmarks show only marginal overheads because they are bound by CPU time spent in user space. As mentioned in Section 7.3, the deficiencies in compile benchmarks apply to the Andrew benchmark as well.

Table IV. Postmark Configurations Used in Our Slowfs Experiments

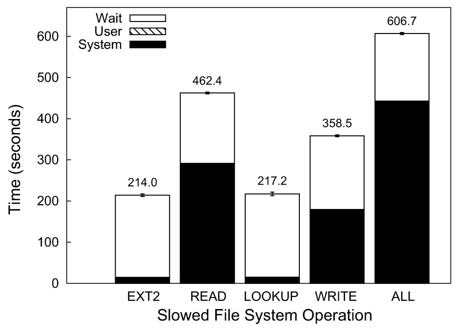
PARAMETER	FSL	CVFS	CVFS-LARGE
Number of Files	20,000	5,000	5,000
Number of Subdirectories	200	50	50
File Sizes	512 bytes–10KB	512 bytes–10KB	512–328,072 bytes
Number of Transactions	200,000	20,000	20,000
Operation Ratios	equal	equal	equal
Read Size	4KB	4KB	4KB
Write Size	4KB	4KB	4KB
Buffered I/O	no	no	no



(a) Postmark: FSL configuration



(b) Postmark: CVFS configuration



(c) Postmark: CVFS-LARGE configuration

Fig. 8. Time taken to execute the Postmark benchmark with several configurations while slowing down various file system operations using Slowfs. Note the different scales for the *y*-axes.

Postmark. We tested Slowfs with three different Postmark configurations (described in Table IV). The FSL configuration is the one we have been using in our laboratory [Aranya et al. 2004; Wright et al. 2003a], the CVFS configuration is from the CVFS research paper [Soules et al. 2003], and CVFS-LARGE is similar to the CVFS configuration, but we used the median size of a mailbox on our campus's large mail server for the file size. We have used a similar configuration before [Muniswamy-Reddy et al. 2004], but in these experiments we updated the file size. We used Postmark version 1.5, and used Slowfs to slow down each of the operations separately, as well as together, by a factor of four. The results are shown in Figure 8.

The graphs show us two important features of this benchmark. First, if we look at the EXT2 bar in each graph, we can see how much changing the configurations can affect the results. The three are very different, and clearly incomparable (FSL takes over 55 times longer than CVFS, and CVFS-LARGE is still almost twice as long as FSL). Second, we can see that different configurations show the effects of Slowfs in varying degrees.

For example, slowing down the reads yields an elapsed time overhead of 3.6% for FSL (16.7% system time), 14.1% for CVFS (19.4% system time), and 116% for CVFS-LARGE (2,055% system time) over ext2 (EXT2). We can see that in the CVFS configuration, there is no wait time on the graph. This is because the configuration was so small that the benchmark finished before the flushing daemon could write to the disk. CVFS has larger overheads than FSL because writes are a smaller component of the benchmark, and so reads become a larger component. CVFS-LARGE has higher overheads than the other two configurations because it has much larger files, and so there is more data to be read. Similarly, when all operations are slowed down (ALL), there is an elapsed time overhead of 12.3% for FSL (85.6% system time), 65.8% for CVFS (83.5% system time), and 183% for CVFS-LARGE (3,177% system time).

Depending on the characteristics of the file system being tested, it is possible to choose a configuration that will yield low overheads. Even so, we see that Postmark sufficiently exercises the file system and shows us meaningful overheads, so long as the workload is large enough to produce I/O (i.e., the working set is larger than available memory and the benchmark runs for enough time). This is in contrast to the compile benchmarks, which barely show any overheads.

13. CONCLUSIONS

We have examined a range of file system and storage benchmarks and described their positive and negative qualities, with the hope of furthering the understanding of how to choose appropriate benchmarks for performance evaluations. We have done this by surveying 106 file-system and storage-related research papers from a selection of recent conferences and by conducting our own experiments. We also provided recommendations for how benchmarks should preferably be run and how results might be presented. This advice was summarized in our suggested guidelines (see Section 3).

We recommend that with the current set of available benchmarks, an accurate method of conveying a file or storage system's performance is by using at least one macrobenchmark or trace, as well as several microbenchmarks. Macrobenchmarks and traces are intended to give an overall idea of how the system might perform under some workload. If traces are used, then special care should be taken with regard to how they are captured and replayed, and how closely they resemble the intended real-world workload. In addition, microbenchmarks can be used to help understand the system's performance, to test multiple operations to provide a sense of overall performance, or to highlight interesting features about the system (such as cases where it performs particularly well or poor).

Performance evaluations should improve the descriptions of *what* the authors did, as well as *why* they did it, which is equally important. Explaining the reasoning behind one's actions is an important principle in research, but is not being followed consistently in the fields of file system and storage performance evaluations. Ideally, there should be some analysis of the system's expected behavior, and various benchmarks either proving or disproving these hypotheses. Such analysis may offer more insight into a behavior than just a graph or table.

We believe that the current state of performance evaluations, as seen in the surveyed research papers, has much room for improvement. Computer science is still a relatively young field, and the experimental evaluations need to move further in the direction of precise science. One part of the solution is that standards clearly need to be raised and defined. This will have to be done both by reviewers putting more emphasis on a system's evaluation, and by researchers conducting experiments. Another part of the solution is that this information needs to be better disseminated to all. We hope that this article, as well as our continuing work, will help researchers and others to understand the problems that exist with file and storage system benchmarking. The final aspect of the solution to this problem is creating standardized benchmarks, or benchmarking suites, based on open discussion among file system and storage researchers.

We believe that future storage research can help alleviate the situation by answering questions such as the following.

(1) How can we accurately portray various real-world workloads?
(2) How can we accurately compare results from benchmarks that were run on different machines and systems and at different times?

To help answer the first question, we need a method of determining how close two workloads are to each other. To answer the second, we believe that benchmark results can be normalized for the machine on which they were run. To standardize benchmarks, we believe that there is a need for a group such as the SPC to standardize and maintain storage and file system benchmarks, and for such benchmarks to be more widely used by this community. We are currently working on some of these problems and there is still much work to be done, but we hope that with time the situation will improve.

The project Web site (www.fsl.cs.sunysb.edu/project-fsbench.html) contains the data collected for this survey, our suggestions for proper benchmarking techniques, and the source-code and machine configurations that we used in the experiments throughout the article.

ACKNOWLEDGMENTS

We would like to thank all the people who helped review earlier drafts of this work, as well as attendees of the FAST 2005 BoF for their valuable comments. We would also like to thank the members of the File Systems and Storage Laboratory at Stony Brook for their helpful comments and advice.

REFERENCES

Notes: Entries marked with asterisks are papers that were included in the survey.

*Abd-El-Malek, M., II, W. V. C., Cranor, C., Ganger, G., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J. 2005. Ursa Minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 59–72.

*Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, 1–14.

Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. 2007. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 31–45.

*Aguilera, M. K., Ji, M., Lillibridge, M., MacCormick, J., Oertli, E., Andersen, D., Burrows, M., Mann, T., and Thekkath, C. A. 2003. Block-Level security for network-attached disks. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 159–174.

Akkerman, W. 2002. Strace software home page. `www.liacs.nl/~wichert/strace/`.

*Anderson, D. C., Chase, J. S., and Vahdat, A. M. 2000. Interposed request routing for scalable network storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 259–272.

*Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A. 2002. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 175–188.

*Anderson, E., Kallahalla, M., Uysal, M., and Swaminathan, R. 2004. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 45–58.

Andrerson, D. 2002. Fstress: A flexible network file service benchmark. Tech. Rep. TR-2001-2002, Duke University. May.

*Aranya, A., Wright, C. P., and Zadok, E. 2004. Tracefs: A file system to trace them all. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 129–143.

*Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Burnett, N. C., Denehy, T. E., Engle, T. J., Gunawi, H. S., Nugent, J. A., and Popovici, F. I. 2003. Transforming policies into mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (ACM SIGOPS)*, Bolton Landing, NY, 90–105.

Blaze, M. 1992. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter Conference*, San Francisco, CA.

Bray, T. 1996. Bonnie home page. `www.textuality.com/bonnie`.

Bryant, R., Forester, R., and Hawkes, J. 2002. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Monterey, CA, 259–274.

Bryant, R., Raddatz, D., and Sunshine, R. 2001. PenguinoMeter: A new file-I/O benchmark for Linux. In *Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, CA, 5–10.

Chen, P. M. and Patterson, D. A. 1993. A new approach to I/O performance evaluation—Self-Scaling I/O benchmarks, predicted I/O performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (ACM SIGOPS)*, Seattle, WA, 1–12.

*Cipar, J., Corner, M. D., and Berger, E. D. 2007. TFS: A transparent file system for contributory storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 215–229.

Coker, R. 2001. Bonnie++ home page. `www.coker.com.au/bonnie++`.

*Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., and Sankar, S. 2004. Row-Diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 1–14.

*Dabek, F., Kaashoek, M. F., Karger, D., and Morris, R. 2001. Wide-Area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada.

*DeBergalis, M., Corbett, P., Kleiman, S., Lent, A., Noveck, D., Talpey, T., and Wittle, M. 2003. The direct access file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 175–188.

*Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2002. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, Monterey, CA, 177–190.

*Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005. Journal-guided resynchronization for software RAID. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 87–100.

*Dimitrijevic, Z., Rangaswami, R., and Chang, E. 2003. Design and implementation of semi-preemptible IO. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 145–158.

*Eisler, M., Corbett, P., Kazar, M., Nydick, D. S., and Wagner, J. C. 2007. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 139–152.

Ellard, D., Ledlie, J., Malkani, P., and Seltzer, M. 2003. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA.

Ellard, D. and Seltzer, M. 2003a. New NFS tracing tools and techniques for system analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, San Diego, CA.

Ellard, D. and Seltzer, M. 2003b. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, San Antonio, TX, 101–114.

*Flinn, J., Sinnamohideen, S., Tolia, N., and Satyanaryanan, M. 2003. Data staging on untrusted surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 15–28.

*Fraser, K. and Chang, F. 2003. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the Annual USENIX Technical Conference*. San Antonio, TX, 325–338.

*Fu, K., Kaashoek, M. F., and Mazières, D. 2000. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 181–196.

*Ghemawat, S., Gobioff, H., and Leung, S. T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles. (ACM SIGOPS)*, Bolton Landing, NY, 29–43.

*Gniady, C., Butt, A. R., and Hu, Y. C. 2004. Program-Counter-Based pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (ACM SIGOPS)*, San Francisco, CA, 395–408.

*Gopal, B. and Manber, U. 1999. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. (*ACM SIGOPS*), New Orleans, LA, 265–278.

*Grönvall, B., Westerlund, A., and Pink, S. 1999. The design of a multicast-based distributed file system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (ACM SIGOPS)*, New Orleans, LA, 251–264.

*Gulati, A., Naik, M., and Tewari, R. 2007. Nache: Design and implementation of a caching proxy for nfsv4. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 199–214.

*Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6,* 1 (Feb.), 51–81.

*Huang, H., Hung, W., and Shin, K. 2005. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 263–276.

HUANG, L. AND CHIUEH, T. 2001. Charm: An I/O-driven execution strategy for high-performance transaction processing. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 275–288.

*JOGLEKAR, A., KOUNAVIS, M. E., AND BERRY, F. L. 2005. A scalable and high performance software iSCSI implementation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 267–280.

JOUKOV, N., TRAEGER, A., IYER, R., WRIGHT, C. P., AND ZADOK, E. 2006. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (ACM SIGOPS)*, Seattle, WA, 89–102.

*JOUKOV, N., WONG, T., AND ZADOK, E. 2005. Accurate and efficient replaying of file system traces. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 337–350.

*KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. 2003. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 29–42.

KAMINSKY, M., SAVVIDES, G., MAZIERES, D., AND KAASHOEK, M. F. 2003. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (ACM SIGOPS)*, Bolton Landing, NY.

KATCHER, J. 1997. PostMark: A new filesystem benchmark. Tech. Rep. TR3022, Network Appliance. www.netapp.com/tech_library/3022.html.

*KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. 2000. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 119–134.

*KIM, M., COX, L., AND NOBLE, B. 2002. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Monterey, CA.

*KROEGER, T. M. AND LONG, D. D. E. 2001. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 105–118.

LASS. 2006. UMass trace repository. http://traces.cs.umass.edu.

*LEE, Y., LEUNG, K., AND SATYANARAYANAN, M. 1999. Operation-Based update propagation in a mobile file system. In *Proceedings of the Annual USENIX Technical Conference*, Monterey, CA, 43–56.

*LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. 2004. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 121–136.

*LU, C., ALVAREZ, G. A., AND WILKES, J. 2002. Aqueduct: Online data migration with performance guarantees. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA.

*LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. 2003. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 131–144.

*LUMB, C. R., SCHINDLER, J., AND GANGER, G. R. 2002. Freeblock scheduling outside of disk firmware. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 275–288.

*MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C., AND ZHOU, L. 2004. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 105–120.

*MAGOUTIS, K., ADDETIA, S., FEDOROVA, A., AND SELTZER, M. I. 2003. Making the most out of direct-access network attached storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 189–202.

*MAGOUTIS, K., ADDETIA, S., FEDOROVA, A., SELTZER, M. I., CHASE, J. S., GALLATIN, A. J., KISLEY, R., WICKREMESINGHE, R. G., AND GABBER, E. 2002. Structure and performance of the direct access file system. In *Proceedings of the Annual USENIX Technical Conference*, Monterey, CA.

*MAZIÉRES, D. 2001. A toolkit for user-level file systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 261–274.

*Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. 1999. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, 124–139.

McDougall, R. and Mauro, J. 2005. FileBench. www.solarisinternals.com/si/tools/filebench/.

*Memik, G., Kandemir, M., and Choudhary, A. 2002. Exploiting inter-file access patterns using multi-collective I/O. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA.

Mesnier, M. P., Wachs, M., Sambasivan, R. R., Lopez, J., Hendricks, J., Ganger, G. R., and O'Hallaron, D. 2007. //TRACE: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 153–167.

*Miller, E., Freeman, W., Long, D., and Reed, B. 2002. Strong security for network-attached storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 1–13.

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

Mummert, L. and Satyanarayanan, M. 1994. Long term distributed file reference tracing: Implementation and experience. Tech. Rep. CMU-CS-94-213, Carnegie Mellon University, Pittsburgh, Pennsylvania.

*Muniswamy-Reddy, K., Holland, D. A., Braun, U., and Seltzer, M. 2006. Provenance-Aware storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 43–56.

*Muniswamy-Reddy, K., Wright, C. P., Himmer, A., and Zadok, E. 2004. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 115–128.

*Muthitacharoen, A., Chen, B., and Mazieres, D. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada.

*Muthitacharoen, A., Morris, R., Gil, T. M., and Che, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, 31–44.

*Ng, W. T., Sun, H., Hillyer, B., Shriver, E., Gabber, E., and Ozden, B. 2002. Obtaining high performance for storage outsourcing. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 145–158.

*Nightingale, E. B., Chen, P., and Flinn, J. 2005. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 191–205.

*Nightingale, E. B. and Flinn, J. 2004. Energy-Efficiency and storage flexibility in the Blue file system. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 363–378.

*Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, 1–14.

*Nugent, J., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. 2003. Controlling your PLACE in the file system with gray-box techniques. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, 311–323.

OSDL. 2004. Iometer project. www.iometer.org/.

OSDL. 2007. Database test suite. www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/.

Ousterhout, J. 1990. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Technical Conference*, Anaheim, CA, 247–256.

Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J. 1985. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, Orcas Island, WA, 15–24.

*Padioleau, Y. and Ridoux, O. 2003. A logic file system. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, 99–112.

*Papathanasiou, A. E. and Scott, M. L. 2004. Energy efficient prefetching and caching. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 255–268.

*Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleinman, S., and Owara, S. 2002. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 117–129.

*Peek, D. and Flinn, J. 2006. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, 219–232.

PEL. 2001. BYU trace distribution center. `http://tds.cs.byu.edu/tds`.

*Peterson, Z. N. J., Burns, R., Ateniese, G., and Bono, S. 2007. Design and implementation of verifiable audit trails for a versioning file system. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 93–106.

*Peterson, Z. N. J., Burns, R., J. Herring, A. S., and Rubin, A. D. 2005. Secure deletion for a versioning file system. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 143–154.

*Prabhakaran, V., Agrawal, N., Bairavasundaram, L. N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005a. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 206–220.

*Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dussea, R. H. 2005b. Analysis and evolution of journaling file systems. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, 105–120.

*Quinlan, S. and Dorward, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 89–101.

*Radkov, P., Yin, L., Goyal, P., Sarkar, P., and Shenoy, P. 2004. A performance comparison of NFS and iSCSI for IP-networked storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 101–114.

*Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. 2003. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 1–14.

Robinson, D. 1999. The advancement of NFS benchmarking: SFS 2.0. In *Proceedings of the 13th USENIX Systems Administration Conference*, Seattle, WA, 175–185.

Roselli, D., Lorch, J. R., and Anderson, T. E. 2000. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, San Diego, CA, 41–54.

Rosenblum, M. 1992. The design and implementation of a log-structured file system. Ph.D. thesis, Electrical Engineering and Computer Sciences, Computer Science Division, University of California.

*Rowstron, A. and Druschel, P. 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada.

Ruemmler, C. and Wilkes, J. 1993. UNIX disk access patterns. In *Proceedings of the Winter USENIX Technical Conference*, San Diego, CA, 405–420.

Ruwart, T. M. 2001. File system performance benchmarks, then, now, and tomorrow. In *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, San Diego, CA.

*Saito, Y., Karamanolis, C., Karlsson, M., and Mahalingam, M. 2002. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, 15–30.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. 1985. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX Technical Conference*, Portland, Oregon, 119–130.

*Santry, D. S., Feeley, M. J., Hutchinson, N. C., Veitch, A. C., Carton, R. W., and Ofir, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, 110–123.

*Sarkar, P., Uttamchandani, S., and Voruganti, K. 2003. Storage over IP: When does hardware support help? In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 231–244.

*Schindler, J., Griffin, J. L., Lumb, C. R., and Ganger, G. R. 2002. Track-Aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 259–274.

*Schindler, J., Schlosser, S. W., Shao, M., and Ailamaki, A. 2004. Atropos: A disk array volume manager for orchestrated use of disks. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 159–172.

*Schlosser, S. W., Schindler, J., Papadomanolakis, S., Shao, M., Ailamaki, A., Faloutsos, C., and Ganger, G. R. 2005. On multidimensional data and modern disks. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 225–238.

Schmidt, A., Waas, F., Kersten, M., Florescu, D., Carey, M. J., Manolescu, I., and Busse, R. 2001. Why and how to benchmark XML databases. *ACM SIGMOD Rec. 30,* 3 (Sept.), 27–32.

*Schmuck, F. and Haskin, R. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 231–244.

*Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the Annual USENIX Technical Conference*, San Diego, CA, 71–84.

Seltzer, M. I., Krinsky, D., Smith, K. A., and Zhang, X. 1999. The case for application-specific benchmarking. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, Rio Rica, AZ, 102–107.

Shein, B., Callahan, M., and Woodbury, P. 1989. NFSSTONE: A network file server performance benchmark. In *Proceedings of the Summer USENIX Technical Conference*, Baltimore, MD, 269–275.

Shepler, S. 2005. NFS version 4. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA. `http://mediacast.sun.com/share/shepler/20050414_usenix_ext.pdf`.

*Shrira, L. and Xu, H. 2006. Thresher: An efficient storage manager for copy-on-write snapshots. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 57–70.

*Sivathanu, G., Sundararaman, S., and Zadok, E. 2006. Type-Safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, 15–28.

*Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2004a. Life or death at block-level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 379–394.

*Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005. Database-Aware semantically-smart storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 239–252.

*Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2004b. Improving storage system availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 15–30.

*Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2003. Semantically-Smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 73–88.

Small, C., Ghosh, N., Saleeb, H., Seltzer, M., and Smith, K. 1997. Does systems research measure up? Tech. Rep. TR-16-97, Harvard University. November.

Smith, K. A. and Seltzer, M. I. 1997. File system aging—Increasing the relevance of file system benchmarks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, 203–213.

SNIA. 2007. SNIA—Storage network industry association: IOTTA repository. `http://iotta.snia.org`.

*Sobti, S., Garg, N., Zhang, C., Yu, X., Krishnamurthy, A., and Wang, R. 2002. PersonalRAID: Mobile storage for distributed and disconnected computers. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 159–174.

*Soules, C. A. N., Goodson, G. R., Strunk, J. D., and Ganger, G. R. 2003. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 43–58.

Spadavecchia, J. and Zadok, E. 2002. Enhancing NFS cross-administrative domain access. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Monterey, CA, 181–194.

SPC. 2007. Storage performance council. `www.storageperformance.org`.

SPEC. 2001. SPEC SFS97_R1 V3.0. `www.spec.org/sfs97r1`.

SPEC. 2003. SPEC SMT97. www.spec.org/osg/smt97/.

SPEC. 2004. SPEC SDM Suite. www.spec.org/osg/sdm91/.

SPEC. 2005a. The SPEC organization. www.spec.org/.

SPEC. 2005b. SPECweb99. www.spec.org/web99.

SPEC. 2007. SPECviewperf 9. www.spec.org/gpc/opc.static/vp9info.html.

*Stein, C. A., Howard, J. H., and Seltzer, M. I. 2001. Unifying file system protection. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 79–90.

*Strunk, J. D., Goodson, G. R., Scheinholtz, M. L., Soules, C. A. N., and Ganger, G. R. 2000. Self-Securing storage: Protecting data in compromised systems. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 165–180.

Swartz, K. L. 1996. The brave little toaster meets Usenet. In *Proceedings of the 10th USENIX System Administration Conference (LISA)*, Chicago, IL, 161–170.

*Tan, Y., Wong, T., Strunk, J. D., and Ganger, G. R. 2005. Comparison-Based file server verification. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, 121–133.

Tang, D. 1995. Benchmarking filesystems. Tech. Rep. TR-19-95, Harvard University.

Tang, D. and Seltzer, M. 1994. Lies, damned lies, and file system benchmarks. Tech. Rep. TR-34-94, Harvard University. December. In *VINO: The 1994 Fall Harvest*.

*Thereska, E., Schindler, J., Bucy, J., Salmon, B., Lumb, C. R., and Ganger, G. R. 2004. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 213–226.

*Tian, L., Feng, D., Jiang, H., Zhou, K., Zeng, L., Chen, J., Wang, Z., and Song, Z. 2007. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 277–290.

*Tolia, N., Harkes, J., Kozuch, M., and Satyanarayanan, M. 2004. Integrating portable and distributed storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 227–238.

*Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Bressoud, T., and Perrig, A. 2003. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, 127–140.

TPC. 2005. Transaction processing performance council. www.tpc.org.

Tridgell, A. 1999. Dbench-3.03 README. http://samba.org/ftp/tridge/dbench/README.

Van Meter, R. 1997. Observing the effects of multi-zone disks. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, 19–30.

*Van Meter, R. and Gao, M. 2000. Latency management in storage systems. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 103–118.

*Veeraraghavan, K., Myrick, A., and Flinn, J. 2007. Cobalt: Separating content distribution from authorization in distributed file systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 231–244.

VERITAS Software. 1999. VERITAS file server edition performance brief: A PostMark 1.11 benchmark comparison. Tech. Rep., Veritas Software Corporation. June. http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf.

VeriTest. 2002. NetBench. www.veritest.com/benchmarks/netbench/.

*Vilayannur, M., Nath, P., and Sivasubramaniam, A. 2005. Providing tunable consistency for a parallel file store. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 17–30.

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

*Wachs, M., Abd-El-Malek, M., Thereska, E., and Ganger, G. R. 2007. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 61–76.

*Wang, A. A., Reiher, P., Popek, G. J., and Kuenning, G. H. 2002. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the Annual USENIX Technical Conference*, Monterey, CA, 15–28.

*Wang, R. Y., Anderson, T. E., and Patterson, D. A. 1999. Virtual log based file systems for a programmable disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 29–44.

*Wang, Y. and Merchant, A. 2007. Proportional-Share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 47–60.

Watson, A. and Nelson, B. 1992. LADDIS: A multi-vendor and vendor-neutral SPEC NFS benchmark. In *Proceedings of the 6th USENIX Systems Administration Conference (LISA VI)*, Long Beach, CA, 17–32.

*Weddle, C., Oldham, M., Qian, J., Wang, A. A., Reiher, P., and Kuenning, G. 2007. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 245–260.

*Weil, S., Brandt, S., Miller, E., Long, D., and Maltzahn, C. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, 307–320.

Wittle, M. and Keith, B. E. 1993. LADDIS: The next generation in NFS file server benchmarking. In *Proceedings of the Summer USENIX Technical Conference*, Cincinnati, OH, 111–128.

Wright, C. P., Dave, J., and Zadok, E. 2003a. Cryptographic file systems performance: What you don't know can hurt you. In *Proceedings of the 2nd IEEE International Security In Storage Workshop*. IEEE Computer Society, Washington, DC, 47–61.

Wright, C. P., Joukov, N., Kulkarni, D., Miretskiy, Y., and Zadok, E. 2005. Auto-Pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Anaheim, CA, 175–187.

*Wright, C. P., Martino, M., and Zadok, E. 2003b. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, 197–210.

*Yu, X., Gum, B., Chen, Y., Wang, R. Y., Li, K., Krishnamurthy, A., and Anderson, T. E. 2000. Trading capacity for performance. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, 243–258.

*Yumerefendi, A. R. and Chase, J. S. 2007. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, San Jose, CA, 77–92.

*Zadok, E. 2002. Overhauling Amd for the '00s: A case study of GNU autotools. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Monterey, CA, 287–297.

*Zadok, E., Anderson, J. M., Bădulescu, I., and Nieh, J. 2001. Fast indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 289–304.

*Zadok, E., Bădulescu, I., and Shender, A. 1999. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, Monterey, CA, 57–70.

*Zadok, E. and Nieh, J. 2000. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, San Diego, CA, 55–70.

*Zhang, C., Yu, X., Krishnamurthy, A., and Wang, R. Y. 2002. Configuring and scheduling an eager-writing disk array for a transaction processing workload. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 289–304.

*Zhang, Z. and Ghose, K. 2003. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 59–72.

*Zhou, Y., Philbin, J., and Li, K. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, 91–104.

*Zhu, N., Chen, J., and Chiueh, T. 2005a. TBBT: Scalable and accurate trace replay for file server evaluation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 323–336.

*Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K., and Wilkes, J. 2005b. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 177–190.

Received May 2007; revised November 2007; accepted November 2007