Benchmarking Filesystems

Benchmarking Filesystems

Diane Tang

TR-19-95

Benchmarking Filesystems

A Thesis presented

by

Diane L. Tang

to Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 10, 1995

Abstract

One of the most widely researched areas in operating systems is filesystem design, implementation, and performance. Almost all of the research involves reporting performance numbers gathered from a variety of different benchmarks. The problem with such results is that existing filesystem benchmarks are inadequate, suffering from problems ranging from not scaling with advancing technology to not measuring the filesystem.

A new approach to filesystem benchmarking is presented here. This methodology is designed both to help system designers understand and improve existing systems and to help users decide which filesystem to buy or run. For usability, the benchmark is separated into two parts: a suite of micro-benchmarks, which is actually run on the filesystem, and a workload characterizer. The results from the two separate parts can be combined to predict the performance of the filesystem on the workload.

The purpose for this separation of functionality is two-fold. First, many system designers would like their filesystem to perform well under diverse workloads: by characterizing the workload independently, the designers can better understand what is required of the filesystem. The micro-benchmarks tell the designer what needs to be improved while the workload characterizer tells the designer whether that improvement will affect filesystem performance under that workload. This separation also helps users trying to decide which system to run or buy, who may not be able to run their workload on all systems under consideration, and therefore need this separation.

The implementation of this methodology does not suffer from many of the problems seen in existing benchmarks: it scales with technology, it is tightly specified, and it helps system designers. This benchmark's only drawbacks are that it does not accurately predict the performance of a filesystem on a workload, thus limiting its applicability: it is useful to system designers, but not for users trying to decide which system to buy. The belief is that the general approach will work, given additional time to manipulate the prediction algorithm.

CHAPTER 1

Introduction and Background

1.1 Motivation

If the number of papers in an area is an indication of research interest, then filesystems research is one of the most interesting research topics in operating systems today. Looking in three recent operating system conference proceedings: OSDI (November, 1994), USENIX (June, 1994), and SOSP (December, 1993), the sixty-eight papers published can be sorted in the following manner:

- 17 on file systems (including the distributed file systems papers listed below)
- 15 on distributed systems (including the distributed file systems and distributed shared memory papers listed below)
- 13 on memory systems (including the distributed shared memory systems papers below)
- 8 on performance issues
- 7 on distributed file systems
- 6 on distributed shared memory
- 5 on mobile computing
- 5 on security

While approximately one out of every four papers published in operating systems relates to filesystems, less than half that amount (eight) relate to performance issues. This is surprising since almost all of the papers use performance numbers to back the claims made.

Whether these filesystem papers examine a modification to an existing filesystem, or a design and implementation for a new filesystem, performance is the crucial issue. The researchers are trying to show that their ideas yield better performance than the status quo, perhaps only for some particular workload. A variety of different mechanisms are used to “prove” these assertions of higher performance: simulation (especially if the system is not yet implemented), hand-waving (“is this idea not wonderful? but we haven’t made any measurements yet...”), and benchmarks. Benchmarks are typically a suite of programs containing some performance gathering mechanism run at user level that yield results used to compare different systems. Benchmarks are also the most commonly used method for determining performance improvements.

For example, one current debate in filesystem research concerns which filesystem design is “the best”: the Fast File System (FFS) or the Log-structured File System (LFS) [5][9]. Researchers argue whether the gains in LFS are due mostly to the asynchronous writes or to the layout of data and meta-data on disk, and whether FFS, when augmented with clustered reads and writes, is competitive [10]. The benchmarks used in this debate include a modified version of the Andrew benchmark, TPC-B, and a suite of micro-benchmarks (defined and discussed in the next section) [2][3].

The benchmarks used in filesystem research papers have several problems. First, there is no standard benchmark, whereas in processor design research SPECint92 and SPECfp92 predominate. In filesystems, the closest to a standard is the Andrew benchmark, but even then, some researchers use the original version while others use a modified version, such as Ousterhout’s [2][8]. Many researchers also write their own benchmarks: In the original LFS paper, Rosenblum wrote a suite of micro-benchmarks [9]. When Seltzer tried to reproduce the results, she used Rosenblum’s description to write her own version [10]. This lack of standardization, and even of sharing, makes comparing results from different projects and papers difficult, if not impossible.

Secondly, existing benchmarks used to measure filesystems are inadequate, regardless of whether or not they were designed to do so. Problems include not scaling with technology, not measuring the filesystem (or only measuring part of the filesystem), and not yielding useful results, *i.e.*, results that help a user determine how a system might perform on a different workload or that point a system designer towards possible areas for improvement.

This thesis focuses on determining what functionality is required of a filesystem benchmark, and then defining such a benchmark. The rest of this chapter presents background information needed throughout the thesis. Chapter 2 examines existing benchmarks, while chapter 3 lays out the criteria by which to judge a filesystem benchmark and the functionality required. Chapter 4 presents the proposed benchmarking methodology and an implementation, dtangbm, while chapter 5 presents an example of using the benchmark. Chapter 6 concludes the thesis, and points out some directions for future work.

1.2 Different Types of Benchmarks

Benchmarks may be categorized in two ways. One way is to categorize a benchmark as being either a synthetic or an application benchmark; the other way is as a macro- or micro- benchmark.

Application benchmarks consist of programs and utilities that a user can actually use. For example, SPECint92 consists of six applications including a C compiler, a lisp interpreter, and a spreadsheet [1]. Synthetic benchmarks, on the other hand, model a workload by executing various operations in a mix consistent with the target workload; the NFS benchmarks (nfsstone, nhfsstone, and LADDIS) allow the user to input a target mix of operations, *i.e.*, what percentage of the workload should consist of create's, getattr's, *etc*. [6][7].

Synthetic benchmarks are more flexible than application benchmarks: They usually have a larger number of parameters that might allow them to scale better with technology and to increase the number of different workloads they can model. However, the problem with synthetic benchmarks is that they do not measure any real work. This makes their results questionable because the operations completed in a synthetic benchmark might not take the same amount of time in a real application. Either the synthetic benchmark might add overhead that does not exist in a real application, or a real application might incur overhead not modeled in the benchmark. There is no answer to this question, although conventional wisdom so far disregards the problem.

The second way to categorize a benchmark is as a macro-benchmark or a micro-benchmark. Macro-benchmarks measure the entire system, and usually model some workload; they can be either synthetic or application benchmarks. Micro-benchmarks measure a specific part of the system: They can be thought of as a subset of synthetic benchmarks in that they are artificial; however, they do not try to model any real workload whatsoever. An example of a micro-benchmark is the create micro-benchmark from the original LFS paper: It timed how long the system took to create 10,000 files [9].

Micro-benchmarks have two major problems. First, it is easy to distort results using micro-benchmarks. Because there is no standard suite of micro-benchmarks, many researchers write their own set to show how they improved this one aspect of system performance. What is not shown is whether this improvement detracts from other aspects of the system's performance. The other main problem with micro-benchmarks is that they neither complete real work nor do they model a real workload: A mix of operations will result in different behavior than the same operation repeated over and over again.

Micro-benchmarks are, however, excellent for pointing out potential areas for improvement within the system.

1.3 Why Benchmarks are Used

Not only are there different types of benchmarks, but there are different reasons to use benchmarks, each reason having different requirements. In 1972, Lucas stated that the three reasons to obtain performance numbers are selection evaluation ("which system is best for me"), performance monitoring ("how can I tweak the system to improve performance"), and performance projection ("how well will this idea for a system perform") [4]. He further states that benchmarks are excellent for selection evaluation, adequate for performance monitoring, and insufficient for performance projection.

Looking at Lucas's statements, there are really two audiences with different requirements for benchmarks. One audience consists of customers looking to buy a system; what they care about is which system will perform best under their workload. It is this audience that has spawned benchmarks such as IOStone (see Chapter 2), which only yield one number as a final result. This type of benchmark is fairly useless, because only customers whose workload at least approximates the benchmark's target workload can use the result, and then only for relative comparisons.

System designers comprise the other audience; they use benchmarks to point them towards possible areas for improvement either in the current system or in the design of a new system; a benchmark that yields only one number is of no use to this audience of benchmarkers.

In general, customers need macro-benchmarks, while system designers need a combination of macro- and micro-benchmarks. Customers probably prefer application benchmarks, because matching workloads is easier. System designers probably tend to prefer synthetic benchmarks because of the greater control and flexibility.

1.4 References

[1] B. Case, "Updated SPEC Benchmarks Released: SPEC92, New Multiprocessor Benchmarks Now Available", Microprocessor Report, September 16, 1992, 14-19.

[2] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West, "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems, February 1988, 51-81.

[3] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc. 1992.

[4] H. C. Lucas, Jr., "Performance Evaluation and Monitoring", Computing Surveys, September 1972, 79-91.

[5] M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", ACM Transactions on Computer Systems, August, 1984, 191-197.

[6] M. K. Molloy. "Anatomy of the NHFSSTONES Benchmark", Performance Evaluation Review 19, 4 (1992).

[7] B. Nelson, B. Lyon, M. Wittle, B.Keith, "LADDIS – A Multi-Vendor and Vendor-Neutral NFS Benchmark", UniForum Conference, January 1992.

[8] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" Proceedings of the 1990 USENIX Summer Technical Conference, June 1990, 247-256.

[9] M. Rosenblum, J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM Transactions on Computer Systems, February 1992, 26-52.

[10] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, V. Padmanabhan, "File System Logging versus Clustering: A Performance Evaluation", Proceeding of the 1995 USENIX Technical Conference, 249-264.

CHAPTER 2

Existing Benchmarks: The Good, the Bad, and the Ugly

Given the different ways to categorize benchmarks and the different audiences that use benchmarks, this chapter analyzes existing benchmarks to help future benchmark designers decide what to do and what not to do. For each benchmark, the purpose (what it is supposed to measure) is compared to what is actually measured.

Before the actual analysis of the benchmarks, the system configuration and the utilities used to determine what the benchmarks actually measure are described.

2.1 System Configuration

The testbed used for analyzing the benchmarks is the machine, *sake*. While running the benchmarks and monitoring utilities, *sake* is in single-user mode to minimize extraneous activity. *Sake*'s configuration is described in Table 2.1, while its disks are described in Table 2.2. Note that for the Seagate disk, two times are quoted for seek times in Table 2.2. This difference is due to the fact that for reads, the head does not need to be as close to the surface as for writes. Also, note that even though there are multiple disks on a machine, there is only a single SCSI controller.

Sake has a fast processor, so no benchmark that measures the filesystem should be entirely CPU bound. Similarly, the main disk being used, the Seagate ST12550N, is relatively fast given today's technology, and so should not be a bottle-neck except for I/O-intensive applications. *Sake* also has enough virtual memory that no benchmark should have to page extensively, if at all. The size of the buffer cache is not that large, especially in comparison to machines that allow the buffer cache to grow dynamically, so stressing the cache should not be difficult for benchmarks to do.

System Name	Processor	Operating System	Main Memory / Buffer Cache	Block Size (Bytes)	Disks	Maximum Bus Bandwidth
sake	Pentium	BSDI 1.1	40 MB / 4 MB	8192	sd0: Seagate ST12550N sd1: DEC RZ25L	5 MB / s

Table 2.1 **Sake's Configurations**

Model Number	Capacity (Formatted)	Cache Size	Track-to-track Seek	Average Seek	Maximum Seek	Spindle Speed
Seagate: ST12550N	2,139 MB	1,024 KB	0.6 ms / 0.9 ms	8 ms / 9 ms	17 ms / 19 ms	7200 rpm
DEC: RZ25L	535 MB	256 KB	2 ms	11 ms	24 ms	5400 rpm

Table 2.2 **Disk Statistics**

2.2 Examining Benchmarks

Three unix utilities are used to gather statistics about system resources used by the benchmarks: `iostat`, `vmstat`, and `time`. `Iostat` reports statistics about the I/O subsystem, such as the number of sectors transferred per second and milliseconds per seek [4]. `Vmstat` reports statistics about the virtual memory subsystem, including such data as the number of page faults per second and the number of system calls per second [4]. For a full listing of the fields outputted by `iostat` and `vmstat`, see Appendix A. `Time` reports the total elapsed time, the total time spent in user mode for the process, and the total time spent in system mode [5]. `Iostat` and `vmstat` are run concurrently with the benchmark, while `time` is used to insure that the benchmark is not significantly affected by the statistics-gathering utilities (the output of `time` when running the benchmark alone is compared to the output of `time` when running the benchmark concurrently with `iostat` and `vmstat`).

2.3 Benchmarks

Seven benchmarks are analyzed in this chapter: SPECint92, the Andrew File System Benchmark, TPC-B, the Bonnie benchmark, IOStone, the Self-Scaling I/O Benchmark, and the NFS benchmarks (nfsstone, nhfsstone, and LADDIS). Each benchmark was designed with a different purpose in mind, and while they are not all filesystem benchmarks, we can learn a lesson applicable to benchmarking in general from each of them. A brief description of each benchmark and its classifications is given in Table 2.3.

Benchmark	Synthetic	Application	Macro-	Micro-	Brief Description
SPECint92		X	X		A suite of applications designed to measure CPU integer performance.
Andrew		X	X		A suite composed entirely of unix utilities, designed to model the workload generated by system developers.
TPC-B		X	X		A database benchmark modeling a bank.
Bonnie	X			X	A suite consisting of six micro-benchmarks, designed to model I/O intensive applications.
IOStone	X		X		A filesystem benchmark modeling a workload based on various filesystem analyses.
Self-Scaling	X		X	X	An I/O subsystem benchmark that parameterizes the I/O subsystem into 5 parameters, measuring and predicting performance in terms of those five parameters. It is difficult to classify this benchmark as either a macro- or micro-benchmark, since it gives information both on how the I/O subsystem as a whole would perform, and on specific aspects of the I/O subsystem.
NFS	X		X		A benchmark designed to measure NFS server performance by modeling a workload based on an input mix of operations.

Table 2.3 **Benchmarks: A brief description and classification**

2.3.1 SPECint92

In 1992, the System Performance Evaluation Cooperative (SPEC) released two new benchmark suites, one to measure CPU integer performance (SPECint92) and one to measure CPU floating point performance (SPECfp92). These two benchmark suites replace the original, much-criticized benchmark released in 1989 and are designed to measure the relative speed of a computer system on CPU-intensive floating point or integer applications. The applications are intended to be computationally intensive, rather than I/O, graphics, or network intensive [7].

SPECint92 and SPECfp92 numbers are published by manufacturers of processors, such as HP, DEC, and Sun. These numbers are used by consumers to help them decide which system to buy, and by researchers to help determine what technology (superscalar, super-pipelined, *etc*.) yield the best performance at this time. Due to the difficulty in obtaining a FORTRAN compiler for BSDI, we examine only SPECint92 here. The different applications that compose this suite are listed in Table 2.4. Figure 1 through Figure 6 are graphs and discussions of system usage resulting from running each of the six applications three times (*i.e.*, there are three plots per graph, one plot per run). As stated before, this information is obtained by running `iostat` and `vmstat` simultaneously with the benchmark. Note that in all of these runs, all of the executables, input files, output files, and statistics gathered from `iostat` and `vmstat` are on the Seagate disk of *sake*.

Application	Description
espresso	Minimizes Boolean functions. Written in C.
li	A lisp interpreter, with an input program that solves the 9-queens problem. The interpreter is written in C, and the input program in lisp.
eqntott	Translates a boolean equation into a truth table. Written in C.
compress	Performs data compression on a 1 MB file using adaptive Lempel-Ziv coding. Written in C.
sc	Performs computations within a UNIX spreadsheet. Written in C.
gcc	Consists of the GNU C compiler converting preprocessed files into optimized Sun-3 assembly code. Written in C.

***Table 2.4* Applications used in SPECint92**

Many people using SPEC question whether SPECint92 measures only (or even mainly) the CPU, or whether other factors, such as paging activity or I/O requests, might affect the final results [9]. Given that many people spent a great deal of time to develop and market the SPEC benchmark suite, it comes as no surprise that SPECint92 does in fact measure CPU performance. All six applications are CPU bound with over 90% of their time spent in completing user CPU operations. There is very little paging activity: almost all of the paging that does occur is demand paging, and gcc is the only application with other paging activity.

While half of the applications do not use the disk much at all, those that do are still CPU bound. The reason for this apparent contradiction is that the disk activity observed on *sake*, under FFS, consists mainly of asynchronous writes to disk: Eqntott creates one output file that is 13 MB in size, compress creates two 1 MB files, and gcc creates 76 files totalling almost 6 MB in size. Note that under other operating systems such as DOS, however, all writes are synchronous and therefore I/O operations would affect SPECint92 performance more.

SPECint92's results are based on the time needed to run all six benchmarks. Since the benchmarks are CPU bound, and since other system resources used, such as the I/O subsystem, do not significantly affect the results, SPECint92 is a good measure of CPU integer performance. However, a good CPU benchmark does not usually also yield a good filesystem benchmark: the majority of the filesystem calls made are asynchronous writes that do not stress the system, and therefore do not reflect filesystem performance. The other downside to SPECint92, as a benchmark, is that by itself, it does not really help a user decide which system to buy. Many workloads consist of several processes running concurrently; in such a situation, the CPU is not the sole factor determining performance: The operating system plays a crucial role as well.

Figure 1. SPECint92 statistics, Espresso: (a) shows that not only is espresso CPU bound, but mostly user CPU bound. The increases in system CPU usage in (a) match the spikes in (b) and correspond to the reading in and writing out of the four input files and four output files (during the course of one run, espresso is executed four times, once for each input file). The peak seek times shown in (c) are the average seek time of the Seagate disk.

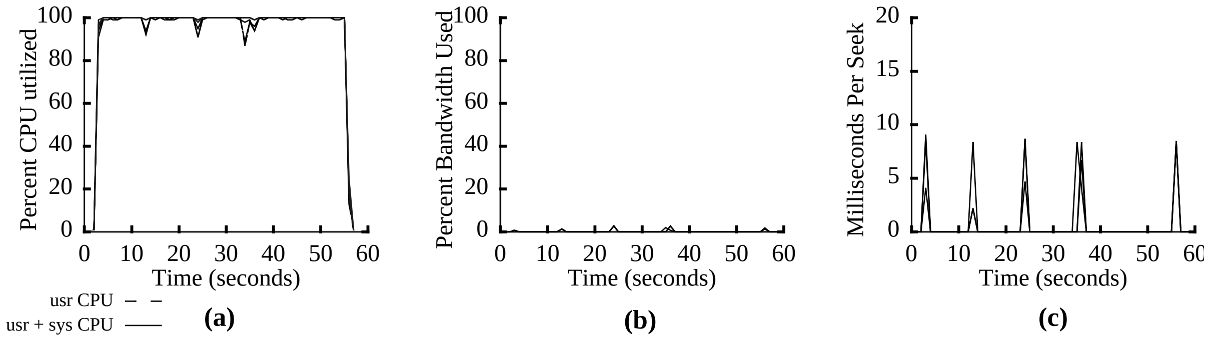


Figure 2. SPECint92 statistics, Li: Like espresso, (a) shows that li is mostly user CPU bound. The slightly more substantial system CPU usage is due to the fact that its output is written one character at a time using `putc()`. The small spikes in (b) and (c) correspond to when one block of the output file is written to disk (in BSDI-FFS, data is written to disk asynchronously once a full block has been written). Once again, the peak seek times in (c) correspond to the average seek time of the Seagate disk.

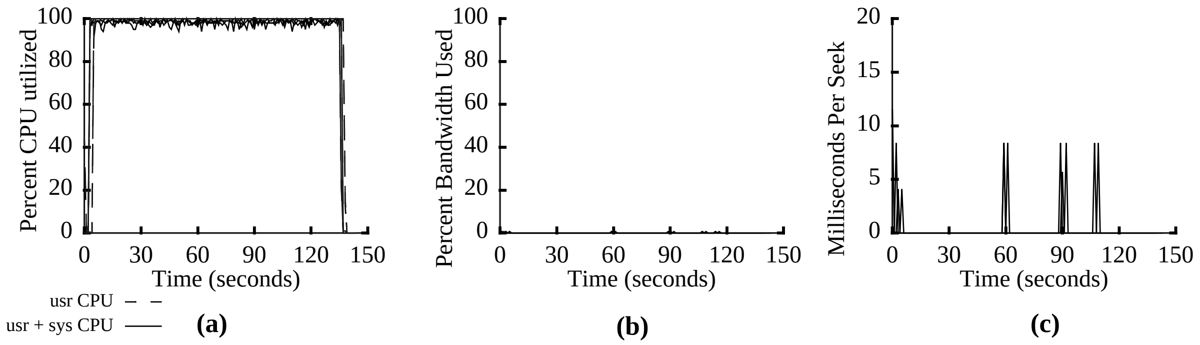


Figure 3. SPECint92 statistics, Eqntott: (a) shows that eqntott is mostly user CPU bound. The system CPU usage is attributable to the system calls used to write the 13 MByte output file and corresponds to the disk utilization in (b) and (c). Despite the substantial disk utilization, eqntott is still CPU bound, demonstrating how asynchronous I/O operations do not seriously affect CPU performance.

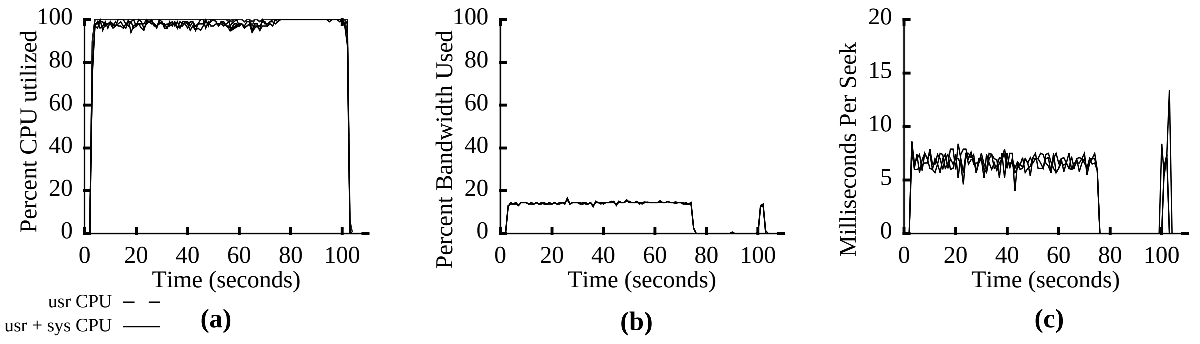


Figure 4. SPECint92 statistics, Compress: Like the previous benchmarks, compress is CPU bound, with system CPU usage attributable to dealing with the output files (two 1 MB files). As in eqntott, the substantial disk utilization in (b) does not affect CPU performance severely due to the asynchronicity of the writes. (The lack of activity for the first two and last two seconds are due to the gathering of statistics before and after the benchmark execution, showing the system's steady state. This steady state exists in all of the graphs, however, due to the short execution time of compress, this state seems emphasized)

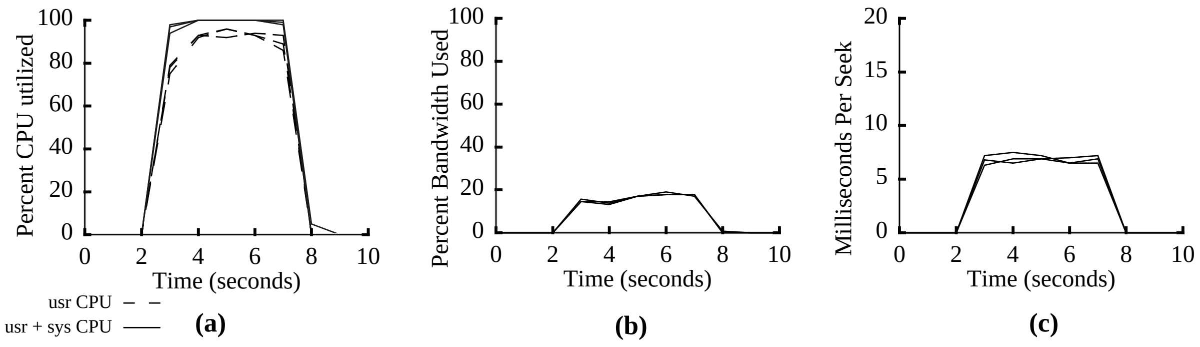


Figure 5. SPECint92 statistics, Sc: From (a), we can see that sc is CPU bound; the higher system CPU usage is attributable to the writes to the screen, which are redirected to files on disk. The seek times in (c) reflect that the files are fairly close together, and thus the average seek time of slightly less than 9 ms.

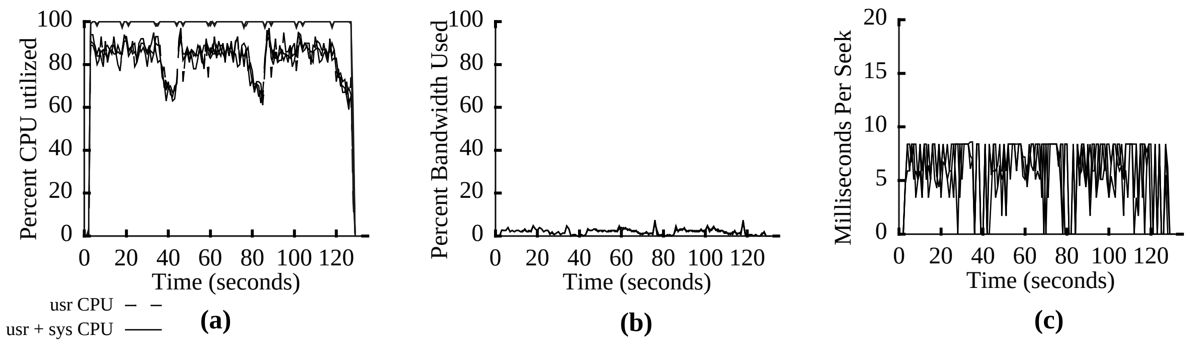
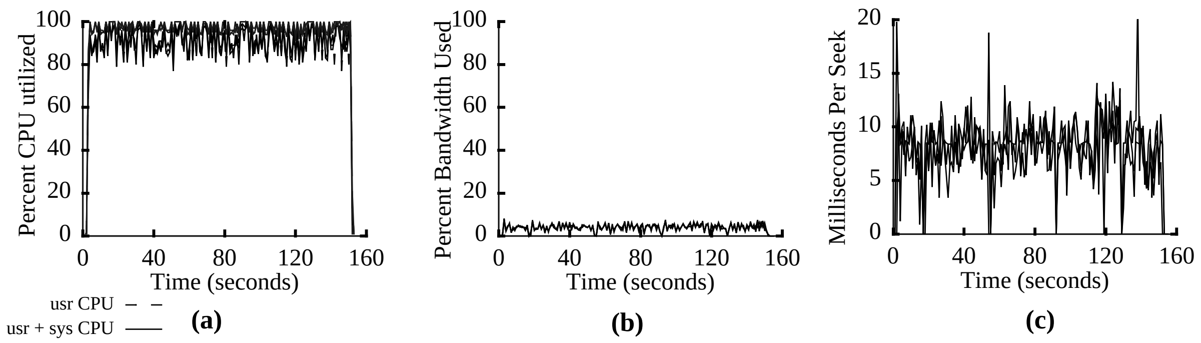


Figure 6. SPECint92 statistics, Gcc: Gcc is essentially CPU bound (a) despite the significant disk utilization in (b). Like eqntott and compress, these numbers are due to the composition of the disk operations, consisting mainly of asynchronous writes. The seek times average at 9 ms; the higher variance reflects the usage of many files, most, but not all, of which are close together due to the layout policy of FFS [14].



2.3.2 TPC-B

In 1988, a group of companies joined together to form the Transaction-processing Performance Council (TPC). The purpose of TPC was to standardize a benchmark for use in the data-processing industry; before, benchmarks had been proposed (*e.g.*, DebitCredit, TP1 [1]), but were sufficiently imprecise that different vendors could use loopholes to improve their own performance rating. In November of 1989, TPC released TPC-A, which is a more tightly specified version of the DebitCredit benchmark [1]. The following August, they released TPC-B, which is based on TP1. Essentially, TPC-B is TPC-A without the terminal interactions; TPC-B is a pure database benchmark, whereas TPC-A measures on-line transaction processing [10].

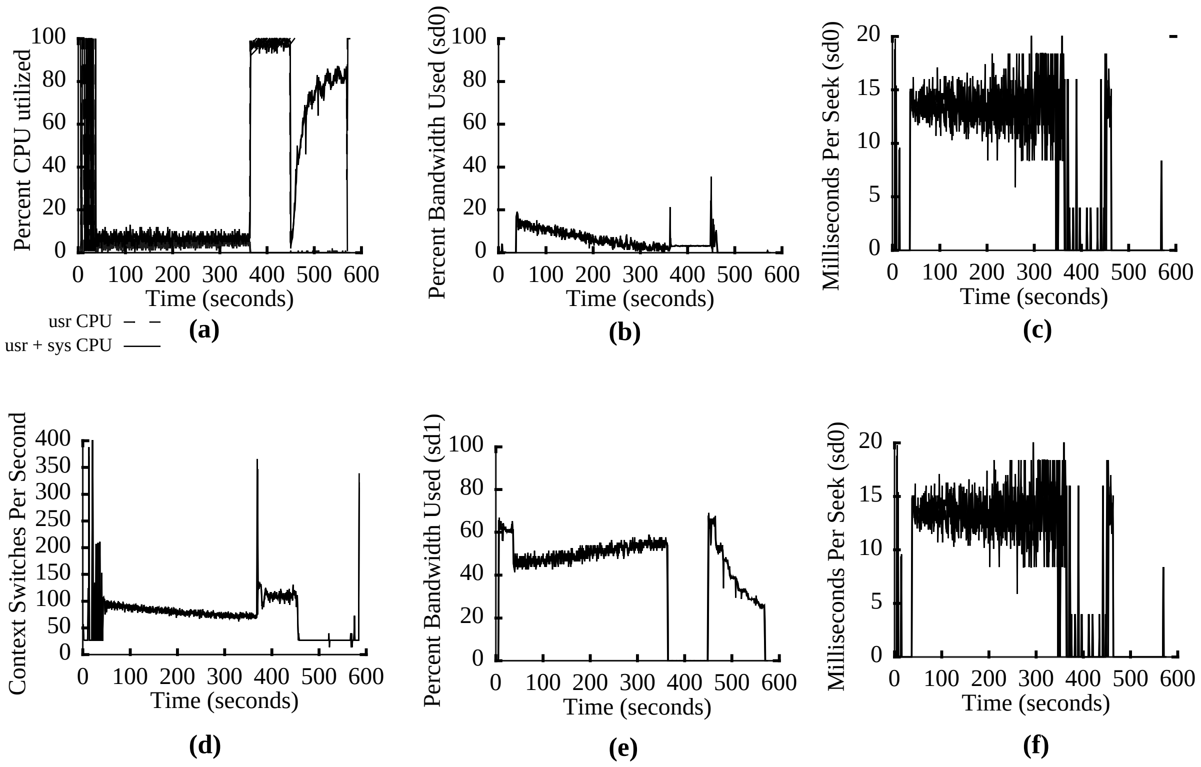
TPC-B simulates a bank using four databases: one for the transactions to accounts, one for transactions made by a teller, one for transactions made in a branch, and one for logging all transactions. Each transaction (withdrawal or deposit) requires that the account, branch, and teller databases be updated to reflect the new account balance, and that a history record be appended to the history file. The implementation used here is from Seltzer's work on support for transaction processing in file systems [19].

TPC-B was run three times, using a database with 100,000 account records, 100 teller records, 10 branch records, and 100,000 history records, totalling approximately 25 MBytes of data. Each run completed 20,000 transactions. The databases and error log were on sd0 (Seagate), while the log file (8 MBytes), all the shared memory files (20 MBytes) and lock files were on sd1 (DEC). Figure 7 shows the results from those runs.

While TPC-B has problems as a database benchmark, only problems relating to its suitability as a filesystem benchmark are discussed here. First, TPC-B does not measure all filesystem functionality; it is a data benchmark with essentially no meta-data operations: all of the files are opened or `mmap`'d once at the beginning and closed at the end. Even as a data benchmark, TPC-B is not complete, measuring only random access patterns, mostly read-modify-write. While this makes sense for a database benchmark in which random records in already existing databases are modified, it means that TPC-B is useful only to those with such a workload, typically database users. However, given that TPC-B is not a very good database benchmark, perhaps it is not useful for this group either [13].

Despite its faults, TPC-B scales well, simply by increasing the size of the databases and the number of transactions. Also, by presenting performance as a function of this load, a user can predict how the system would perform under his workload (same access patterns but different load) better than if TPC-B just gave a single number as the final result.

Figure 7. TPC-B Results: A comparison between the CPU utilization graph, (a), to the disk utilization graphs, (b) and (c), shows the alternation between disk usage and CPU usage. Disk utilization on sd0, (b), which has the databases, is not very high; the access pattern on sd0 is random, which can be determined from the high seek times in (c). Disk utilization is higher on sd1, which contains the log file and shared memory files; by comparing graphs (e) and (f) to Figure 11 (b) and (c), it can be seen that the utilization is high given the random access pattern determined from (f). From the context switches in (d) in comparison to Figure 10 (b), it can be seen that the data access pattern is mostly a read-modify-write pattern.



2.3.3 Andrew

The Andrew Benchmark, developed in 1988 at CMU, was designed to compare the performance of the Andrew File System (AFS) to that of other distributed filesystems. It was supposed to model the workload system designers would generate, but it was not designed to be a representative workload for benchmarking purposes; rather, it was designed to provide a way at the time to compare the performance of different distributed filesystems, especially the penalty paid for remote accesses to a server [11].

Andrew consists of the following phases:

1. Creating a file directory hierarchy

2. Copying files to that hierarchy
3. Examining the new copy (`stat`) (Note that this phase reads meta-data.)
4. Reading the new copy (`grep`) (Note that this phase reads data.)
5. Compiling the copy

Figure 8 shows the statistics from running `iostat` and `vmstat` concurrently with the benchmark. Matching the graphs with the output from Andrew itself, the following observations can be made:

- After the copy, the entire hierarchy is in the buffer cache. The rise in system CPU utilization corresponds to the increase in the number of system calls completed, showing that all accesses to files after they are copied into the hierarchy are satisfied in the cache.
- The compile phase of the benchmark takes almost two-thirds of the running time. During this phase, the benchmark is CPU bound.

Despite the fact that the Andrew Benchmark was not designed as an all-purpose filesystem benchmark, it is being used as such, and therefore it will be critiqued as such.

There are two main problems with Andrew. First, Andrew has not scaled with technology: its fixed size data set is too small; it might have stressed the system when it was first developed, but it no longer does so. Secondly, Andrew is almost entirely CPU bound rather than filesystem bound. A kernel build with a two-level hierarchy totalling approximately 10 MBytes has roughly the same characteristics as the compile phase in Andrew, and a look back at gcc from the SPECint92 suite shows that it is even more CPU bound. The kernel build graphs (Figure 9) show burstier disk traffic, probably due to the synchronous reads necessary to read in the source and include files from disk; these synchronous reads are satisfied by the cache in Andrew. Compiles are CPU bound because the majority of the disk operations consist of asynchronous writes to output files; as a result, although compiles are typical of system designers, they do not make a very good component for filesystem benchmarks.

Andrew’s output consists of how long each phase takes to complete. Phase 1 measures meta-data write throughput; Phase 2 measures both data and meta-data read and write throughput with no distinction between the four. Phase 3 consists of meta-data reads, which are satisfied in the attribute and buffer cache, while Phase 4 consists mainly of data reads, also satisfied in the buffer cache. The final phase consists of data reads and writes; the reads are satisfied in the cache, while the writes are asynchronous. By breaking up the output into phases and presenting the time for each phase, the benchmarker can get an idea of how well different parts of the file system perform, such as the buffer cache bandwidth, minimum size of the buffer cache, and throughput for meta-data writes. However, because several phases consist of more than one type of operation and because there is no way to determine what percentage of time is spent doing what, these phases do not yield much information at all. Also, because a system’s workload usually consists of either several users, or one user doing several different things at once, this benchmark is not very good for predicting the performance of a filesystem under a real workload, even one consisting of system development. Modifications to Andrew do exist to simulate a multi-user environment; however, there is no standard.

Figure 8. The Andrew Benchmark Statistics: First, note that the five phases are distinct and shaded according to the time needed to complete them. (a) shows that for the first two phases, creating the hierarchy and copying the files, the benchmark must wait for the disk to complete the synchronous meta-data operations. This corresponds to the higher disk utilization seen in (d). Notice in (c) that the number of context switches increases over this period, reflecting that the process is indeed waiting for the I/O operations to complete. The next two phases show high system CPU utilization and an increase in the number of system calls in (b): the system calls are completed rapidly, implying that most of the file accesses in these two phases are satisfied in the cache. The compile phase is dominated by user CPU usage; the substantial disk utilization reflects the asynchronicity of the disk operations. The seek time in (c) also reflects disk utilization: in the beginning, seek time is high as the benchmark reads from one directory and writes to another. The lull in the middle corresponds to the phases when the accesses are satisfied by the buffer cache, and the compile phase averages 9 ms per seek.

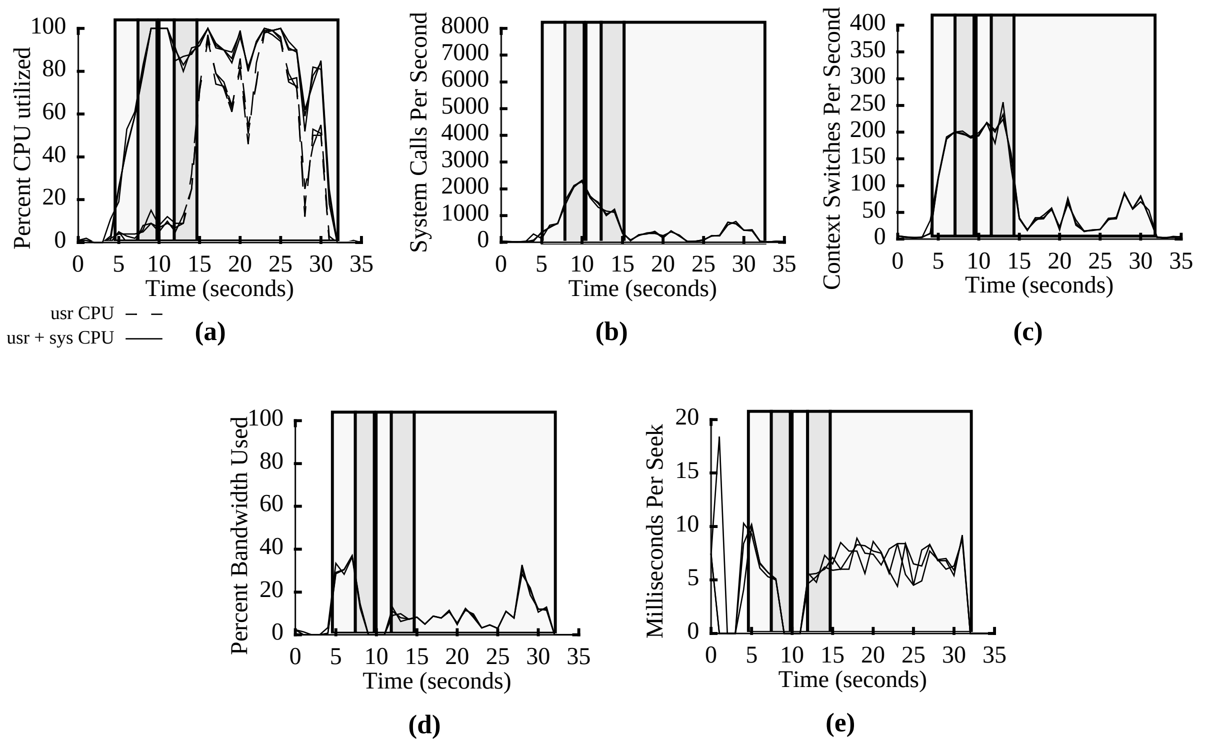
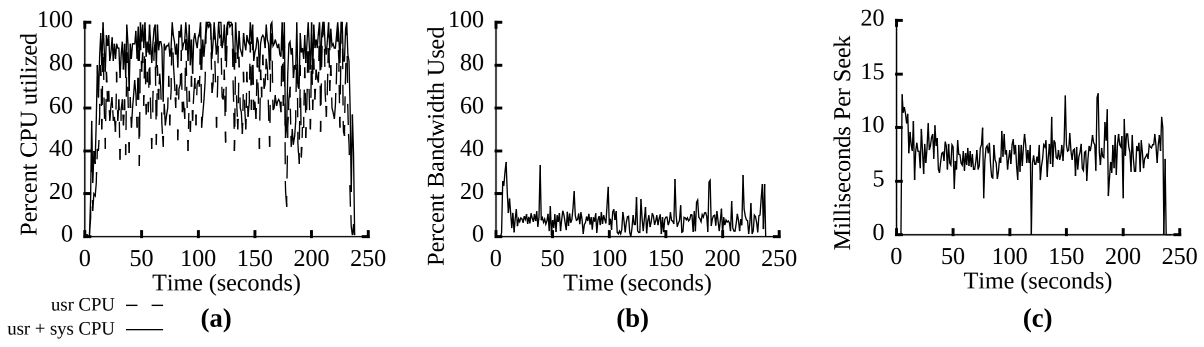


Figure 9. Kernel Build Statistics: (a) shows that a compile is essentially CPU bound, with a fairly high percentage of system CPU usage to handle writing the output files. Note that despite the substantial disk utilization in (b), the build is still CPU bound, reflecting the asynchronicity of the disk operations. The seek time averages at 9 ms.



2.3.4 Bonnie

The Bonnie Benchmark, written by Tim Bray in 1990, is designed to measure bottlenecks in the filesystem. Bonnie's workload is based on file-system activity that was observed to have caused bottlenecks in I/O intensive applications, specifically those found in the text database work done in connection with the New Oxford English Dictionary Project at the University of Waterloo [6].

The Bonnie benchmark consists of six tests; each test consists of a loop small enough to fit into almost any instruction cache, so that there is no paging or swapping during the benchmark's run. The tests are:

1. Create a new file, open a stream associated with the file, write to the file via the stream one character at a time (using `putc()`), and then close the file (using `fclose()` on the stream). Note that by using `fopen()` and `fclose()` data is buffered in stdio, *i.e.*, no data is sent to the filesystem until an entire block has been written.
2. Open the file created in test 1, read a block of the file (`read()`), dirty one word within the block (using a round-robin algorithm), seek back to the beginning of the block (`lseek()`), write the block back (`write()`), and repeat this read-seek-write sequence for the entire file, before closing the file (`close()`).
3. Recreate the file (`open()`), write to the file one block at a time (`write()`), and then close the file (`close()`).
4. Open the existing file, open a stream associated with the file, read the file one character at a time (`getc()`), and then close the stream (`fclose()`).
5. Open the existing file, read the file one block at a time (`read()`), and then close the file (`close()`).
6. Spawn several children. Each child opens the file, seeks to a random spot (`lseek()`), reads one block from that place in the file, dirties that block 10% of the time and writes it back to disk (`write()`). The parent tells the children when to stop.

The user can choose both the size of the file, which defaults to 100 MBytes, and the number of children, which defaults to three. The user should be careful to choose a file size large enough so that the entire file will not fit into the buffer cache, thus testing disk access as well as cache access.

Figure 10 shows the statistics from running `iostat` and `vmstat` concurrently with Bonnie on *sake*'s Seagate disk and Figure 11 shows the statistics on *sake*'s DEC disk.

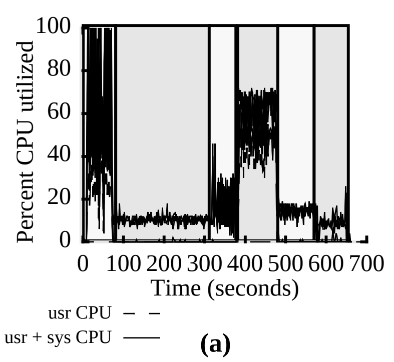
Bonnie is essentially a disk benchmark, finding the peak read and write throughput the file system can provide to the disk, and how long the disk takes to complete a random seek. Tests 1, 2, 3, 4, and 5 all test throughput and layout: how fast I/O requests can be processed by the filesystem and the disk, and how well files are laid out. Tests 1 and 4 also test whether a data buffer exists somewhere in the system, whether it be in the filesystem or in stdio, so that only blocks of data are read from or written to the filesystem or disk. On DOS, for example, every write is synchronous, and no data is buffered anywhere. Tests 4 and 5 also check whether or not the filesystem can detect a sequential read and start reading ahead data. Test 6 measures average seek time, with three processes seeking to random spots in the file and therefore on the disk as well, since the file is large enough to be spread out over at least part of the disk.

System	Per-char throughput KB/s (%CPU util)	Block throughput KB/s (%CPU util)	Rewrite KB/s (%CPU util)	Per-char input KB/s (%CPU util)	Block input KB/s (%CPU util.)	Random Seek (%CPU util.)
sake (Seagate)	931 (45.5%)	937 (9.9%)	422 (6.2%)	1082 (50.6%)	1080 (10.1%)	60.0 sec (5.7%)
sake (DEC)	457 (22.4%)	457 (4.0%)	271 (3.7%)	499 (27.2%)	500 (4.5%)	41.1 sec (4.0%)
champagne	813 (99.5%)	1534 (38.6%)	377 (19.4%)	766 (93.1%)	1766 (46.6%)	42.9 sec (13.3%)
pinot	1972 (61.4%)	1962 (18.0%)	1136 (7.0%)	2676 (88.3%)	2777 (8.7%)	76.6 sec (3.5%)

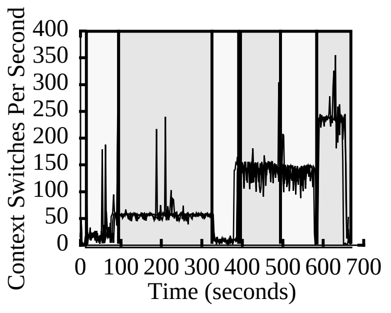
***Table 2.5* Results from Running Bonnie on Different Platforms. The first two rows present the results from running Bonnie on *sake*'s two disks. *Champagne* is a Sun LX running SunOS 4.1.3, and *pinot* is an Alpha AXP running OSF/1.**

While Bonnie is excellent for measuring peak read and write throughput to disk and discovering whether the filesystem has features such as a buffer cache, sequential layout of files, and readahead, Bonnie is not a good filesystem benchmark, lacking any tests of filesystem meta-data performance, for example. Despite this major failing, Bonnie redeems itself by not only measuring what it does well, but by also presenting the results clearly. Each number measures one aspect of the system, and only that one aspect, rather than muddling several different aspects together into one number (Table 2.5 shows the results from running Bonnie on different systems). For example, the similar ratios between the per-character throughput and block throughput on *sake* and *pinot* shows that reads and writes are buffered one block at a time, whereas on *champagne*, the large differential between the two probably reflects the use of clustered reads and writes. That interleave (rotdelay) on *sake* is not optimized for reading or writing and that readahead probably does occur can be inferred from the fact that read throughput is only slightly higher than write throughput. On the other hand, *pinot* probably has interleave set to optimize reading. The different ratios for the rewrite test on the DEC and Seagate disk on *sake* reflect the different disk speeds: The DEC disk, at 5400 rpm, can handle the read and write within two rotations; the Seagate disk, however, rotates fast enough that it sometimes loses a rotation between the read and write due to the time needed by the operating system to handle the interrupt. The different CPU usage between the DEC and the Seagate disk is attributable to the different rotation speeds on the disk; the DEC disk, at 5400 rpm, needs more time to complete each request, and so the loops that generate the requests do not loop as fast, leading to lower CPU usage.

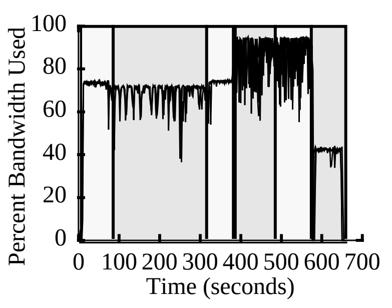
Figure 10. Bonnie Statistics (sake, sd0): All the graphs show a clear demarcation between at least 5 of the 6 tests. (a) shows the high CPU utilization in Tests 1 and 4 due to the looping needed to write or read one character at a time. Despite this high CPU utilization, the disk utilization and throughput are high in (c), due to data being buffered one block at a time. System CPU usage dominates the CPU utilization, needed to process Bonnie's data requests. The asynchronicity of the writes and the corresponding synchronicity of the reads are reflected in the number of context switches in (b), which are really low for writing in Tests 1 and 3, high for reading in Tests 4 and 5, and middling for Test 2, which consists of reads mixed in with writes. The bursts in seek time in (d) reflect how a rotation is often lost between the read and write in Test 2.



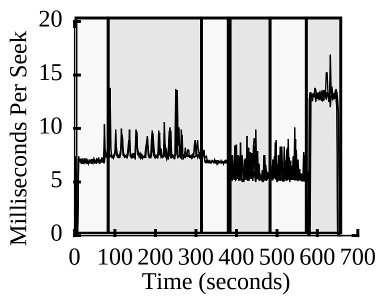
(a)



(b)

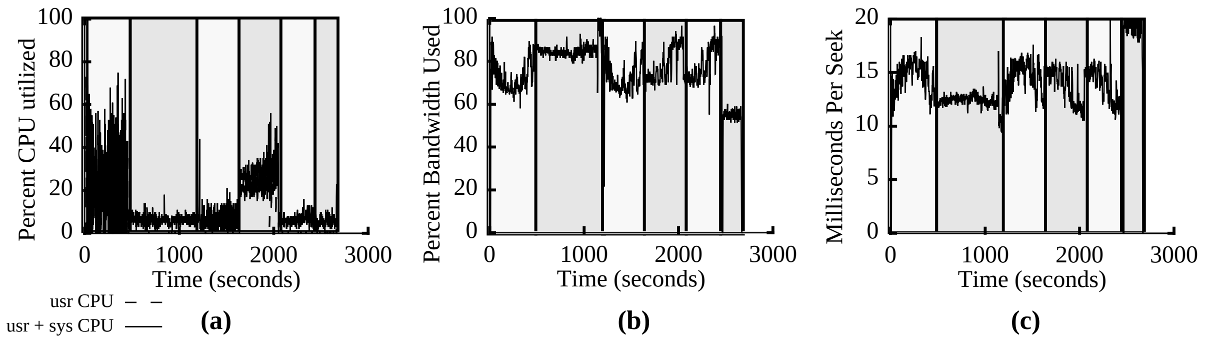


(c)



(d)

Figure 11. Bonnie Statistics (sake, sd1): Comparing the Seagate disk in Figure 10 to the DEC disk here, the DEC disk is noticeably slower (5400 versus 7200 rpm) and probably more fragmented, resulting in poorer file layout. However, the ratio of the times for each test is the same, showing that the main difference is the disk. Notice in (c) that the bursts during Test 2 in Figure 10 are not here, and that Test 2 is relatively shorter for the DEC disk, reflecting how rotations are not lost between the read and the write.



2.3.5 IOStone

IOStone was developed in 1990 by a group of researchers at University of California at Davis to compare the performance of different filesystems, rather than just throughput, *i.e.*, it tries to account for aspects of the filesystem other than the disk, such as disk caches, file system structure, and CPU overhead [18]. The workload modeled by IOStone is based upon several file system analyses [17][12][21][22].

IOStone creates a synthetic filesystem hierarchy, completes a series of I/O requests simulating the locality found in the original analysis of the 4.2 BSD filesystem, and then erases the filesystem [17]. To be more precise, IOStone has three phases:

1. Creates 388 files with sizes ranging from 256 bytes to 64 KBytes and eight spacer files (for a total of 396 files), each one 524 KBytes, placed in the midst of the 388 files. After creating all the files, it reads the 8 spacer files to flush the buffer cache. It then creates a random permutation of all 388 files.
2. Goes through the random permutation, and for each file in that permutation, IOStone opens the file, randomly chooses whether to read or write the file, reads/writes the file in 16 KB blocks (or less if the file were smaller), and then closes the file. It repeats the process three more times, using the same random permutation.
3. Deletes all the files.

The benchmark measures the time needed to complete Phase 2 and returns one number in IOStones/second, calculated by dividing a normalizing constant by the time measured.

We broke IOStone up into 4 phases: the create phase (phase 1), the first pass through the permutation (phase 2a), the next three passes through the permutation (phase 2b), and then the delete phase (phase 3). Figure 12 shows the statistics from `iostat` and `vmstat` during the runs of IOStone on *sake*. The different phases are demarcated by the drops to zero in the graphs.

IOStone has many problems. Like Andrew, IOStone does not scale well, using a fixed-size data set that is too small. As a result, IOStone is more CPU bound than I/O bound: Once the dataset is in the buffer cache, the only disk activity is for asynchronous data writes. Furthermore, reading in the spacer files will not necessarily flush all buffer caches (especially dynamically sized buffer caches); if the buffer cache is not flushed, the final result will reflect this distortion.

Not only does IOStone not scale, but its model is flawed as well. First, IOStone claims to emulate the workload on a typical UNIX workstation, yet it creates a target file system hierarchy that is flat, whereas real file system hierarchies are rarely flat. Next, the workload it measures consists mainly of data operations. On the one hand, a typical workload consists of many meta-data operations as well as data operations [2][3]. On the other hand, given that IOStone only yields one number, at least that number measures mostly (but not only) data operations to minimize the confusion. Also, IOStone uses only one process to access files; most workstations usually have several processes running simultaneously. Finally, while the order in which files are accessed is random, accesses within files are all sequential. The measurement papers on which IOStone is based show that only most access patterns within files are sequential [17][2]. This flawed workload model means that a customer cannot use IOStone to determine how his workload will perform on the system.

One of IOStone's major failings is that it yields only one number, and it is questionable as to what that one number really measures. Phase 2, the only measured phase, consists of data reads and writes and meta-data reads (for the open and close). Reading in the spacer files may flush the buffer cache, but it does not flush the attribute cache (depending on the size of the attribute and name cache), which means that the meta-data reads are satisfied in the cache. Thus, IOStone essentially measures data throughput, dependent in part on file layout. However, because only one number is returned, a system designer has no ideas whether the relatively bad (or good) performance is due to the cache, cache size, disk, file layout, *etc*. Furthermore, the result is very sensitive to buffer cache size. Table 2.6 has IOStone results for different cache sizes (all the numbers are averaged over five runs). Once both the data set and the spacer files fit in the buffer cache, then IOStone performance plateaus, showing the lack of scalability. Also, when the total running time is low, IOStone is not very stable, with a high standard deviation (see Table 2.6). To improve IOStone, the filesystem and workload models should be changed to reflect some semblance of reality, given the purpose for which it was written. It should also be changed so that it scales, thus reducing the areas of instability and increasing the applicability.

Cache Size	IOStones / Sec (Std. Dev.)	Run Time (Std. Dev.)
1 MB	30760 (107.0)	64.02 (0.225)
2 MB	173552 (510.4)	11.52 (0.034)
4 MB	217565 (1449.3)	9.19 (0.061)
6 MB	335220 (1796.3)	5.97 (0.032)
8 MB	337387 (4547.5)	5.93 (0.078)

Table 2.6 **Results from running IOStone on *sake* with different buffer cache sizes.**

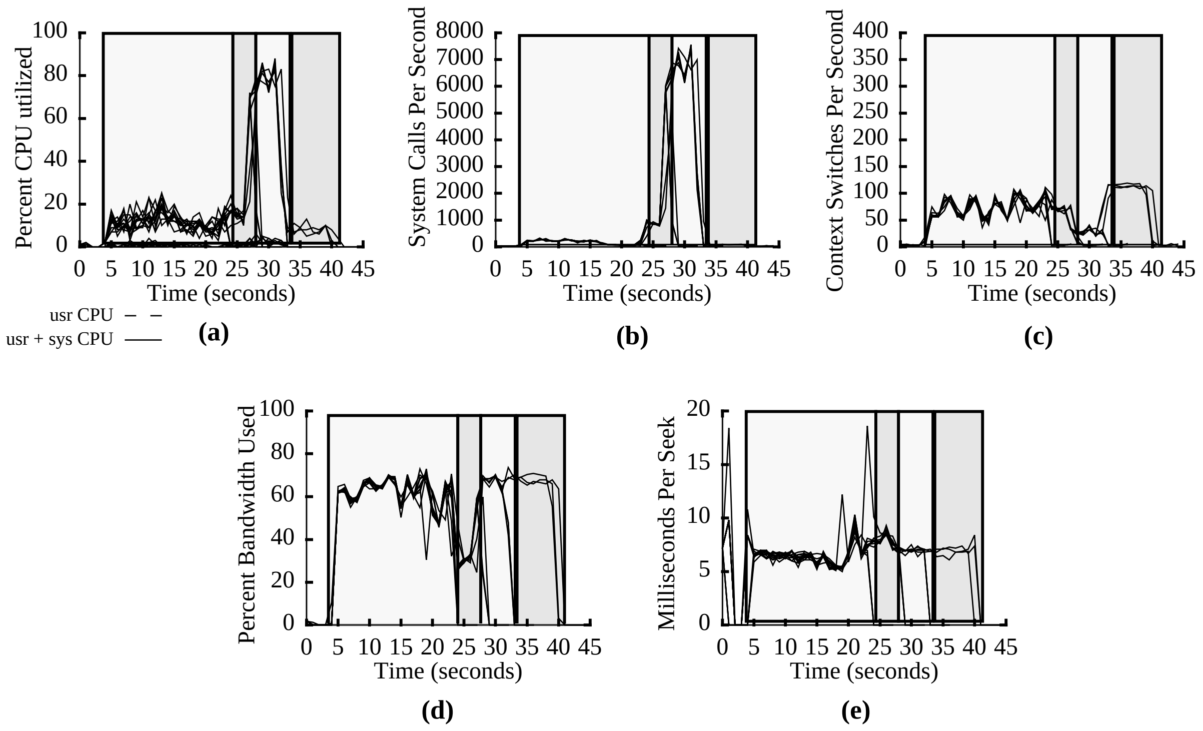


Figure 12. IOStone Statistics: The different phases are demarcated by the shading and the lines to zero. Most of the CPU activity consists of the system fielding file system calls. Note the sharp jump in system CPU usage and number of system calls during Phase 2b, implying that many system calls are being made and completed quickly, which means that the data accesses are fulfilled by the buffer cache. As a result, Phase 2b takes approximately the same amount of time as Phase 2a. Note that the disk utilization (d) increases while the number of context switches (c) decreases in Phase 2b, due to all of the disk operations being asynchronous writes, rather than reads mixed in with writes. We can see that Phase 2a consists of both reads and writes, because the seek time in (e) is slightly higher, implying more collisions and backtracking between different read and write requests. Note that the average seek time during the course of this run is less than 9 ms, implying that the files are not very spread out on the disk.

2.3.6 Pete Chen's Self-Scaling I/O Benchmarks

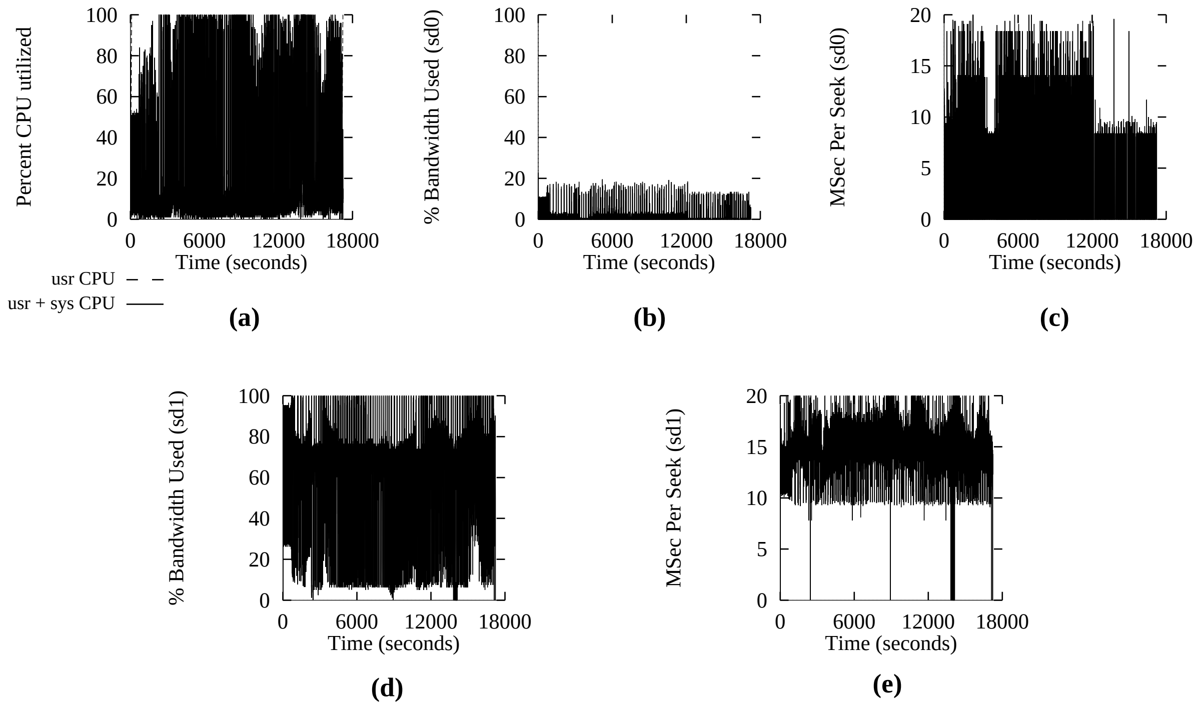
In 1992, Peter M. Chen developed a self-scaling I/O benchmark, designed both to point the system designer towards possible areas for improvement and to scale to measure a wide range of I/O subsystems and workloads while still able to compare the results between systems [8]. His benchmark runs many different workloads on a system, each workload created by varying five parameters: the overall size of the data set, the number of processes running concurrently, the average size of an I/O request (to the nearest block), the percentage of operations that are reads, and the percentage of operations that are sequential. Because the results are presented in terms of these parameters, the benchmarker can examine the results both to determine which area(s) of the system might be improved and to compare the performance of different systems at different parameter values.

The benchmark has two phases. It first finds the *focal vector*, which is the point where each parameter is most stable and far away from any sudden changes in performance. Intuitively, the focal vector is representative of a "typical value," applicable over a wide range of workloads. Once the focal vector has been identified, the benchmark generates five graphs, each plotting throughput as a function of one parameter with the remaining parameters at their focal point value. Using these graphs, Chen introduces the idea of predictive performance. He claims that since the focal vector is generally applicable, the *shape* of a graph should be applicable even when the parameters are not at their focal values. If a workload can be characterized in terms of these five parameters, then the workload performance can be predicted by scaling between the five graphs.

When running this benchmark, the executables were all on *sake*'s Seagate drive, and the test directory was on *sake*'s DEC drive. Both filesystems share the same buffer cache. The graphs in Figure 13 in comparison to the graphs in Figure 11 show that not only does this benchmark stress the target system (sd1), but it also determines how well the system performs at all levels, rather than just peak performance.

This benchmark is scalable, tightly specified, reproducible, descriptive, and prescriptive – for the I/O system. While the filesystem does overlap with the I/O system, this overlap is not sufficient. Its workload model is for the I/O system, with only data operations, rather than for the filesystem, which has data and meta-data operations. As a result, it cannot predict performance for a filesystem workload, and it does not point the system designer towards areas for improvement other than disk scheduling and buffer cache size. By the criteria stated by Chen in his thesis, his benchmark is not a good filesystem benchmark.

Figure 13. The Self-Scaling I/O Benchmark Statistics: The key point to take away from these unclear graphs is that this benchmark does not just test peak performance – it determines I/O subsystem performance for an entire range of workloads instead, thus gathering more data with which to make better predictions.



2.3.7 NFS Benchmarks: nfsstone, nhfsstone, and LADDIS

In 1989, nfsstone was created to measure the performance of Sun NFS servers, due to the popularity of the networked filesystem. It generates a series of NFS requests from a single client to a single server to measure server performance [20]. In 1990, Legato Systems refined nfsstone and called it nhfsstone. The problem with both benchmarks is that they are limited to a single client, and a single client cannot fully stress a server [15]. In 1992, a group of companies joined together to form LADDIS, a benchmark that can stress any NFS server [16][23]. Unfortunately, I have not been able to obtain LADDIS, and I only have nhfsstone version 1.22 from Legato.

This version of nhfsstone has a parent process and several children processes. The parent process spawns and synchronizes several children, collects the final statistics, and checks their consistency. Each child tries to simulate a workload given a target mix of NFS operations and a target average interarrival time between requests. Both the parent and children are run on the same client, and this client is the only client used to test server performance.

By allowing the user to set the number of children, the target mix of calls, and target load, nhfsstone can model many different client workloads. Its major shortcoming is that it was designed to measure server performance rather than client performance, and one client cannot stress a server, no matter how many processes are running on that client. LADDIS improves on nhfsstone by allowing multiple clients, each of which generates a load according to an input mix of operations and an input file access distribution. LADDIS has the potential to be an effective benchmark, scaling in the number of clients and in the load per client, and presenting its results graphically, showing how the performance of a server varies with load. However, LADDIS is limited to NFS servers.

2.4 Conclusion

The analysis presented in this chapter shows that current benchmarks are inadequate for measuring filesystem performance, and that they suffer from the following problems:

- Several benchmarks measure only a subset of the filesystem functionality, typically data throughput. Meta-data operations are usually ignored, even though they constitute a large percentage of the requests made to a filesystem [2][3]. Running `nfswatch`, a utility that monitors what packets are sent to the server, shows that the majority of requests sent to a server are meta-data requests.
- Several benchmarks measure only peak performance, rather than what happens when a real workload is running on a system.
- Most of the benchmarks do not scale with technology to stress today’s systems as well as yesterday’s systems.
- By trying to model a specific workload (*e.g.*, system development, scientific calculation), several benchmarks end up either not modelling any actual workload or being useful to only a very narrow group (while perhaps being widely used).
- A few benchmarks present meaningless results, which cannot be used to predict the performance of a system on realistic workloads or to point system designers towards possible areas for improvement.

Table 2.7 summarizes the flaws of the benchmarks examined in this chapter. The rest of the thesis concentrates on defining what constitutes a “good” filesystem benchmark and defining a benchmark that meets these criteria.

Benchmark	**Not Measuring the Filesystem**	**Peak Performance Only**	**Not Scaling with Technology**	**Lack of General Applicability**	**Meaningless Results**
SPECint92	X	X			
Andrew			X	X	
TPC-B	X			X	
Bonnie	X	X		X	

Table 2.7 **Benchmark Flaw Summary**

Benchmark	Not Measuring the Filesystem	Peak Performance Only	Not Scaling with Technology	Lack of General Applicability	Meaningless Results
IOStone	X		X	X	X
Self-Scaling I/O	X				
NFS benchmarks				X	

Table 2.7 **Benchmark Flaw Summary**

2.5 References

[1] Anon, et al. "A Measure of Transaction Processing Power", Datamation, April 1, 1985, 112.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. "Measurements of a Distributed File System", Proceedings of the 13th Symposium on Operating Systems, 1991, 198-212.

[3] T. Blackwell, J. Harris, M. Seltzer. "Heuristic Cleaning Algorithms for Log-Structured File Systems", Proceedings of the 1995 USENIX Technical Conference, 277-288.

[4] Berkeley Software Distribution. 4.4 BSD System Manager's Reference Manual. California: O'Reilly and Associates, 1994.

[5] Berkeley Software Distribution. 4.4 BSD User's Reference Manual. California: O'Reilly and Associates, 1994.

[6] T. Bray, Bonnie source code, netnews posting, 1990.

[7] B. Case. "Updated SPEC Benchmarks Released: SPEC92m New Multiprocessor Benchmarks Now Available", Microprocessor Report, September 16, 1992, 14-19.

[8] P. M. Chen, D. A. Patterson. "A New Approach to I/O Benchmarks – Adaptive Evaluation, Predicted Performance", UCB/Computer Science Dept. 92/679, University of California at Berkeley, March 1992.

[9] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, A. J. Smith. "Cache Performance of the SPEC92 Benchmark Suite", IEEE Micro, August 1993, 17-26.

[10] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc. 1991.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West. "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems 6, 1 (February 1988), 51-81.

[12] I. Hu. "Measuring File Access Patterns in UNIX", Performance Evaluation Review 14, 2 (1986), 15-20. ACM SIGMETRICS 1986.

[13] C. Levine, J. Gray, S. Kiss, W. Kohler. "Why TPC-A and TPC-B are Obsolete." Draft.

[14] M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry, “A Fast File System for UNIX”, ACM Transactions on Computer Systems, August, 1984, 191-197.

[15] M. K. Molloy. “Anatomy of the NHFSSTONES Benchmark”, Performance Evaluation Review 19, 4 (1992).

[16] B. Nelson, B. Lyon, M. Wittle, B. Keith, “LADDIS – A Multi-Vendor and Vendor-Neutral NFS Benchmark”, UniForum Conference, January 1992.

[17] J. K. Ousterhout, J. DaCosta, et al. “A Trace-Driven Analysis of the UNIX 4.2 BSD File System”, Operating Systems Review 19, 5 (December 1985), 15-24. Proceedings of the 10th Symposium on Operating Systems Principles.

[18] A. Park, J. C. Becker. “IOStone: A Synthetic File System Benchmark”, Computer Architecture News 18, 2 (June 1990), 45-52.

[19]M.I. Seltzer. “File System Performance and Transaction Support”, Memorandum No. UCB/ERL M92/1, January 7, 1992.

[20] Shein, M. Callahan, P. Woodbuy. “NFSStone – A Network File Server Performance Benchmark”, Proceedings of the USENIX Summer Technical Conference 1989, 269-275.

[21] A. J. Smith. “Sequentiality and Prefetching in Database Systems”, ACM Transactions on Database Systems 3, 3 (1978), 223-247.

[22] A. J. Smith. “Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms”, IEEE Transactions on Software Engineering SE-7, No. 4 (1981), 403-417.

[23] M. Wittle, B. E. Keith. “LADDIS: The Next Generation in NFS File Server Benchmarking”, Proceedings of the USENIX Summer Conference, June 1993, 111-128.

CHAPTER 3

Filesystem Benchmark Criteria and Functionality

Chapter 2 focused on why existing benchmarks are inadequate. However, these benchmarks are not all bad. SPECint92 is an example of a benchmark that measures what it claims to measure, and Bonnie is an example of a benchmark that present results clearly to the user without extraneous information and without obscuring the data. The Self-Scaling I/O Benchmark would be an excellent filesystem benchmark, if it measured the filesystem. However, the “goodness” criteria Chen laid out for judging I/O systems can be adapted for filesystems as well [1]. Chen states that an I/O benchmark should be:

- Prescriptive: it should point system designers towards possible areas for improvement.
- I/O bound: the benchmark should measure the I/O system and not, for example, the CPU.
- Scalable with advancing technology.
- Comparable between different systems.
- General: applicable to a wide variety of workloads.
- Tightly specified: no loopholes; clarity in what needs to be reported.

These criteria, with the modification of I/O-bound to filesystem-bound, apply to filesystem benchmarks as well. In fact, these criteria should be applied to most benchmarking methodologies.

The above criteria for judging filesystem benchmarks need one clarification: a definition of filesystem-bound. What needs to be measured? Filesystems may be thought of in terms of the operations a user sees: create, read, write, mkdir, *etc*. Another way of looking at the “big picture” is that the filesystem has two types of operations: data and meta-data. Data operations involve the user’s data, whereas meta-data operations deal with the control structures for the filesystem. In FFS, meta-data consists of inodes, indirect blocks, the free map, and directories.

A third way, perhaps the best for a benchmark writer, is to think in terms of the filesystem's functionality: what is a filesystem responsible for? The following decisions must be made when designing a filesystem:

- What meta-data (filesystem control structures) is there?
- How are blocks allocated to a file? Where is meta-data placed in relation to data? What model of locality is used? For example, FFS tries to have both spatial and logical locality, placing blocks for a single file near each another and placing files in the same directory close together as well.
- How are files named? What algorithm is used for pathname resolution?
- What caching is there, and how are the caches maintained (LRU versus random caching algorithms, determining when data should be flushed to disk)?
- How is disk scheduling done? Can reads and writes be clustered? Can requests be pulled off of the disk queue or be rescheduled?
- How is disk space managed (a free map, blocks versus sectors, *etc*.)? Is there some method used to minimize disk space fragmentation?
- What semantic guarantees are made to the user? For example, when a create system call returns to the user, does the file exist on disk?
- How does a filesystem recover from a crash? Can it recover? How long does it take to recover?
- Can the filesystem handle multiple users, perhaps accessing or even changing the same file?
- How does the filesystem provide protection for a user's data? How well does it work?

The designers of a filesystem need to solve all of these problems, and a filesystem benchmark needs to be able to measure how well their solution works. A filesystem benchmark should also determine what performance gain is due to a clever algorithm in the filesystem versus a clever disk.

The final issue a benchmark writer needs to address concerns which metric to use in evaluating a system. The most common metric is throughput: how many operations can be completed in a a certain amount of time. Throughput is usually expressed in KBytes per second, or IOStones per second, or in general, something per second. Often, however, users care more about latency, *i.e.*, how long it takes to do one operation. The user cares about how long the system takes to respond to a keyboard stroke or to list a directory: "how long to I have to wait?" [2]. Typically, latency is expressed in the average amount of time needed to complete some operation. While throughput and latency are the most common metrics used, they are definitely not the only ones. Other metrics include reliability, security, and efficiency of disk space usage, although throughput and latency are more general.

3.1 References

[1] P. M. Chen, D. A. Patterson. "A New Approach to I/O Benchmarks – Adaptive Evaluation, Predicted Performance", UCB/Computer Science Dept. 92/679, University of California at Berkeley, March 1992.

[2] J. Mogul, "SPECmarks Are Leading Us Astray", The Third Workshop on Workstation Operating Systems, April 1992, 160-163.

CHAPTER 4

An Approach to Benchmarking Filesystems: the dtangbm

Chapter 2 examined current benchmarks to determine what not to do when writing a benchmark, and Chapter 3 laid out the criteria for judging a filesystem benchmark and defined what functionality such a benchmark is responsible for. The rest of the thesis uses this basis to propose a new approach to benchmarking filesystems and present an implementation.

This approach for benchmarking divides the benchmark into two separate parts: a suite of micro-benchmarks, which are run on the filesystem to be tested, and a workload characterizer. Each part generates a set of statistics, one set characterizing the filesystem and one set characterizing the target workload. The performance of the workload on the filesystem can then be predicted using both sets of statistics.

The motivation behind such an approach is to target both system designers and users looking to buy a system. For system designers, the idea is that the micro-benchmarks point out possible areas for improvement, while the workload characterizer tells the designer whether or not the improvement is worth making. An intuitive example is that there is no point in improving `create` performance if `create`'s do not constitute a significant percentage of the workload. Also, the suite of micro-benchmarks should be complete and measure all filesystem functionality so that system designers can determine if an improvement impacts negatively on other aspects of the system.

Users trying to decide which system to buy or install constitute the other audience targeted by this approach. Ideally, they should run their workload on all systems under consideration to decide which system will yield the best performance for them. However, this method is not usually possible, since they probably do not have all the systems on which to test their workload or a benchmark. By separating out the filesystem characterization from the workload characterization in this approach, the results from the micro-benchmarks can be published (similar to SPECint92 and SPECfp92 results), and customers can use the published statistics and the results from the workload characterizer to determine which system is best for them.

The rest of the chapter discusses the approach and the implementation in more detail. Note that dtangbm is the name of the entire benchmark including both the suite of micro-benchmarks and the workload characterizer, while fsbench is the name for the suite of filesystem micro-benchmarks.

4.1 Fsbench: A Suite of Filesystem Micro-benchmarks

Fsbench has four phases: an optional initial measurement phase, a mandatory initial measurement phase, the micro-benchmarks, and the optional extended micro-benchmarks phase for system designers. The optional initial measurement phase measures throughput and seek time for reads and writes to the underlying hardware device. This phase is optional since two filesystems, one with the benchmark executable and one under test, are required. The write tests are optional as well, in case the filesystem under test cannot be overwritten. The mandatory initial measurement phase estimates the sizes of the buffer cache, attribute cache, and name translation cache. These estimates are used to choose file and hierarchy sizes in the next two phases. Next is the main micro-benchmark phase, during which the measurements needed for the comparison to the workload characterizer results are made. The extended micro-benchmark phase follows, with measurements system designers can use but that customers do not need. Table 4.1 gives an overview of fsbench.

Phase	Name	Brief Description
I	Disk Measure-ments	Measures the read and write throughput and seek time to the character (operations do not go through the buffer cache) and block (operations to go through the block device) device that the filesystem under test is on.
II	Cache Sizing	Sizes the buffer cache, attribute cache, and name translation cache
III	Spatial Locality	Data micro-benchmark; measures filesystem block allocation policy to a single file by writing, reading, and overwriting a single file using both random and sequential access patterns.
III	Logical Locality	Data micro-benchmark; measures filesystem block allocation policy to files logically related by being in the same directory by writing, reading, and overwriting the files, accessing the files both in creation order and in random order
III & IV	Metadata Time	Meta-data micro-benchmark; measures common meta-data operations, such as creating and deleting files, making and removing directories, and reading the attributes of both files and directories. Part of the benchmarks belong to Phase III and the other part belong to Phase IV (See Section 4.1.5 and 4.1.6).
IV	Metadata Part 2	Meta-data micro-benchmark; tries to pinpoint possible areas for improvement within meta-data operations more precisely
IV	Readahead	Data micro-benchmark; tries to find degenerate read access patterns to determine if the readahead algorithm in the filesystem behaves properly
IV	Concurrency	Tries to determine which types of operations (data and meta-data, one file versus many files) conflict with one another the most.

Table 4.1 **fsbench Overview**

The purpose behind measuring the disk separately from the filesystem is to help separate out performance gains due to an intelligent disk from performance gains attributable to an intelligent filesystem, while the results from estimating cache sizes are used to start each benchmark from a consistent, known state. The micro-benchmarks are separated into two separate phases to facilitate usability.

Before describing each measurement in detail, three things should be noted. First, fsbench is merely a first cut at a complete suite of micro-benchmarks. Chapter 6 suggests possible enhancements to the suite, including a temporal locality benchmark to match the spatial and logical locality benchmarks. Secondly, fsbench is written using the system call interface in C, and should port to any POSIX-compliant system. The results presented in this thesis are only from BSDI 1.1 and 4.4 BSD, although it has been compiled and tested under SunOS 4.1.3 and OSF/1. Finally, all the numbers presented in this section result from running fsbench on *sake*'s Seagate (sd0) disk unless otherwise stated. See Section 2.1 for the system specifications.

4.1.1 Phase I: Optional Initial Measurements

The purpose of Phase I is to measure the disk so that the results from this phase and Phase III can be compared to determine what overhead the filesystem imposes. This phase is optional because even for the read tests, it requires that the filesystem under test be unmounted, which means that two filesystems, and preferably 2 disks, are needed.

The read measurements are raw disk bandwidth and seek time and disk bandwidth and seek time. The difference between the two is that accesses to the raw, or character, device (*e.g.*, /dev/rsd0a) do not go through the filesystem buffer cache, while the normal, or block, device (*e.g.*, /dev/sd0a) does. While both seek tests determine how long it takes to seek to a random location and read 512 bytes, the raw device bandwidth test reads *N* MB in 64 KB blocks (the current typical maximum transfer size to disk) while the device bandwidth test reads *N* MB in whatever the filesystem block size is (typically 8 KBytes). Here, *N* is the size of the disk as specified by the user (see Appendix B for a complete list of user options); by using the entire disk, this measurement scales with disk features. The measurements made for writes are the same as the measurements made for reads, except that rather than reading from the disk, the tests write to the disk.

To run these tests, the user must customize the scripts the benchmark uses to mount and unmount the filesystem. To run the write tests, the script used to create a new filesystem must be customized as well.

The measurements in Table 4.2 are the results from running only the read tests on *sake*'s Seagate disk (both disks on *sake* contained much needed data, and so the write tests could not be run). Given that the disk rotates at 7200 rpm, and has an average of 35 KBytes per track, it should be able to transfer data at approximately 4 MBytes / sec. Even allowing for operating system interrupt time and assuming that the driver does not allow out-of-order reads, then the expected bandwidth is still 2.7 MBytes / sec – approximately twice the result seen. There is some reason to believe that the Adaptec controller or driver in *sake* may be at fault: substantially higher bandwidth was obtained when an NCR controller and driver was used by Robert Morris (he switched controllers one day “for chuckles”). The substantially lower bandwidth for transfers to the block device is explained by the smaller transfer size; the expected bandwidth, at one 8 KByte transfer per rotation, is 964 KBytes / sec.

Test	**Measurement**
rawDiskReadTime	1304 KBytes / sec
diskReadTime	949 KBytes / sec
rawDiskReadSeekTime	16.46 msec per seek
diskReadSeekTime	21.29 msec per seek

Table 4.2 **Disk Measurements for *sake*'s Seagate disk**

4.1.2 Phase II: Non-optional Initial Measurements

This phase estimates the size of the buffer cache, attribute cache, and name translation cache. These sizes are used to flush the caches and to determine file and hierarchy sizes so that the micro-benchmarks in phases III and IV scale.

To estimate the size of the buffer cache, a binary search algorithm is used. Starting with a file of size 20 MBytes, the file is written and then read sequentially. The time needed to read the last block of the file (which should be in the cache) *n* times is measured, where *n* is the size of the file in filesystem blocks. The time needed to read a random permutation of the blocks in the file is also measured. If those two times are within some skew factor, then the cache is at least the size of the file. Otherwise, if the time to read the random permutation is greater, then the cache must be smaller than the file size. A binary search is used to narrow down the size of the cache (the size is doubled if no maximum bound exists), and the algorithm terminates when the minimum and maximum bounds are within 10 blocks of one another.

The skew factor mentioned above is used to account for possible timing differences in the two accesses measured. The value used equals *n * the timer granularity of* `gettimeofday()`, where *n* is the size of the file in blocks. The time granularity defaults to 10 ms if the environment variable HZ is not set; if it is set, then the timer granularity equals 1 / HZ. Because the point of this measurement is to obtain an upper bound on the cache size, even though `gettimeofday()` is used to measure all *n* accesses, the thought is that if it were used for each of the *n* accesses, each time returned could be off by whatever the timer granularity is. This bound on the timer skew is therefore used as the bound on the access skew time, and each measurement is for *n* accesses, thus *n* times the timer granularity.

This algorithm also works for dynamically sized buffer caches, where the buffer cache and virtual memory are unified main memory. Because this is run on a system in single-user mode (or at least with minimal extraneous activity), the buffer cache can take pages from virtual memory either until only active processes are in virtual memory or until the maximum size of the buffer cache has been reached. Up to that point, all the pages taken from virtual memory are clean, and therefore do not need to be written to disk. This algorithm has been tested on an Alpha running OSF/1, where the initial size of the buffer cache was significantly smaller than its maximum size; the algorithm did indeed find the maximum size of the buffer cache.

To find the size of the attribute cache, the same algorithm is used with two minor differences: rather than using 1 file of size *n* and timing accesses to blocks in that file, *n* distinct files are used, and the time to read the attributes is measured instead. Thus, rather than timing *n* reads of the last block of the file, the time to stat the last file read in is measured instead, and rather than using a random permutation of the *n* blocks in the file, a random permutation of the *n* files is used. Similarly, to find the size of the name translation cache, rather than *n* distinct files, *n* distinct different paths (directories) are used. See Appendix C for the algorithms used to create the hierarchies.

These algorithms were tested on *sake*, with the results presented in Table 4.3. The fact that the estimated size of the attribute cache is much larger than the actual size is not a problem, since the algorithm is aiming for an overestimate anyway.

Cache	Estimated Size	Actual Size
Buffer	3358720 bytes	3317760 bytes
Attribute	2928 inodes	562 inodes
Name Translation	742 names	562 names

Table 4.3 **Cache Size, Estimated and Actual**

Currently, the algorithms used cannot determine if no cache exists. In such a situation, they will currently loop forever. Real operating systems (*e.g.*, not DOS) usually have the caches, so this is currently not a big problem. However, the algorithms should be improved to include this case to increase the benchmark's applicability.

Another problem with sizing the caches is that they are not independent of one another. For one, depending on the filesystem, inodes that are not in the attribute cache may be in the buffer cache and accessed in approximately the same amount of time. This can skew the sizing of the attribute cache, but the problem is solvable by flushing the buffer cache before the two measurements are made. The other major interdependence is between the attribute and name translation cache. If the attribute cache is much larger than the name translation cache, and all the pathnames do not fit in the name cache, then pathname resolution time will skew the sizing of the attribute cache. Similarly, if the name translation cache is much larger than the attribute cache, then inode access time will skew the sizing of the name cache. Because it is more likely that the attribute cache is larger than the name cache, the second problem is not likely to occur, and the first problem will only occur if the attribute cache is several orders of magnitude larger than the name translation cache (fewer directories are used when sizing the attribute cache than when sizing the name translation cache – *n* distinct files, rather than *n* distinct directories).

4.1.3 Phase III: Spatial Locality

The phase following the initial measurement phases is the first set of micro-benchmarks. The first two micro-benchmarks in this phase test block allocation to a single file for sequential and random access patterns. In both benchmarks, three measurements are made: the time needed to initially write to the file (write and block allocation time), the time needed to read the file, and the time needed to overwrite the file. In one benchmark, the file is read and written sequentially, while in the other benchmark, three different random permutations of the blocks in the file are used for the three measurements, thus defeating any system that might save the previous access pattern. The buffer cache is flushed before each measurement taken (see Appendix D for how caches are flushed).

There are four expected results from these two benchmarks:

- The throughputs from the random access benchmark should be lower than the throughputs from the sequential access benchmark.
- The throughputs from the sequential access benchmark should be a high percentage of the disk bandwidths (see Section 4.1.1); if not, the filesystem imposes significant overhead to keep track of a file.
- The write and allocate throughput should be lower than the overwrite throughput, since the filesystem should need extra time to allocate blocks and indirect blocks when initially writing to the file.
- Finally, if the filesystem is optimized for writes, then the write bandwidth should be significantly higher than the read bandwidth; otherwise, if the filesystem is optimized for reading, then the read bandwidth should be significantly higher than the write bandwidth.

The results from running these two benchmarks are in Table 4.4. As expected, the random access pattern bandwidths are lower than the sequential access pattern bandwidths, and the bandwidths from the sequential access pattern are close to the readDiskBandwidth (see Table 4.2). The difference is attributable to the overhead necessary to keep track of the file (inodes, indirect blocks).

Unexpectedly, the sequential read bandwidth is lower than the sequential write bandwidth, but not significantly lower, implying that the filesystem is optimized neither for reading nor writing. From Bonnie, it has already been shown that *sake*'s filesystem does not do aggressive readahead, perhaps due to incorrectly set parameters in the filesystem (such as the rotational delay); even so, read bandwidth was higher than write bandwidth in Bonnie. The other unexpected result is that the write and allocate bandwidth is higher than overwrite bandwidth. The reason for both unexpected results is the increased fragmentation of the filesystem under fsbench in comparison to the fragmentation of the system under Bonnie, due to both the mechanisms used to make space on the disk ("what can I delete?") and the fragmentation resulting from killing ("`kill -9 <pid>`") the benchmark, such as the directory that no one can delete. As a result, each synchronous read takes more time due to the increased seek time and writes, even though asynchronous, need to find, read, and modify the indirect blocks. Note than in Chapter 5, the results from this benchmark turn out as expected.

Access Pattern	Name	Bandwidth (Std Dev.)	% Read Disk Bandwidth
Sequential	Write & Allocate	912 KBytes / sec (0.44)	96%
	Read	887 KBytes / sec (0.95)	93%
	Overwrite	899 KBytes / sec (0.20)	94%
Random	Write & Allocate	847 KBytes / sec (63.6)	89%
	Read	484 KBytes / sec (2.05)	51%
	Overwrite	830 KBytes / sec (68.9)	87%

Table 4.4 **Results from the Spatial Locality Data Benchmarks, averaged over 5 trials.**

4.1.4 Phase III: Logical Locality

The third micro-benchmark is also a data benchmark, testing how blocks are allocated to files logically related to one another, *i.e.*, files in the same directory. In this benchmark, *n* files are created and initially written, where *n* equals $2^{floor(\log bufferCacheSize) - 6}$. These files range in size between 64 bytes and the size of the buffer cache: one file the size of the buffer cache, two files half the size of the buffer cache, and so forth down to 64 bytes. The files are placed in directories according to the algorithm laid out in Appendix C, with the different file sizes distributed randomly among the directories. When actually creating and initially writing to the files, a directory is randomly chosen, and a file in that directory is created and written; in this manner, the creation order of files within a directory is known, but the overall creation order is not. This rather complex method is used to try to avoid unrealistically optimal placement of directories, attributes, and file blocks as might happen if the benchmark just created all of the files in one directory before moving on to the next one.

Once this basis has been created, three measurements are taken: time to open and close all the files, time to open, read, and close all the files, and time to open, write, and then close all of the files. Note that only one file is open at a time for all of the measurements, and that all three caches are flushed before each measurement. Each of these three measurements are taken for three different access patterns: creation order within the directories (which is the same as alphabetical), random order within a directory only, and random order.

The results from running this benchmark on *sake* are in Table 4.5. One result, which at first glance seems unexpected, is that the write bandwidth is lower than the read bandwidth, especially given the asynchronicity of the writes, the fragmentation of the filesystem, and the lack of aggressive readahead on *sake*. However, because one file is not opened until the previous one is closed, the synchronous meta-data read must wait for all of the asynchronous writes queued earlier to complete, thus making many asynchronous writes appear synchronous and yielding a lower final bandwidth.

Access Pattern	Name	Total Time (Std.Dev.)	Measurement	% Read Disk Bandwidth
Creation Order	open-close	649 msec (3.5)		
	read	101890 msec (5.2)	790 KBytes / sec	83%
	write	143857 msec (3.0)	559 KBytes / sec	59%
Random Within Directory	open-close	731 msec (110.2)		
	read	105083 msec (250.1)	767 KBytes / sec	81%
	write	141979 msec (145.2)	566 KBytes / sec	60%
Random	open-close	740 msec (144.7)		
	read	104879 msec (522.0)	768 KBytes / sec	81%
	write	143491 msec (142.2)	560 KBytes / sec	59%

***Table 4.5* Results from the Logical Locality Data Benchmark. The total size of the data set used was 82 MBytes, spread out over 136 files.**

Another unexpected result is that the different access patterns have essentially the same throughput, with only the write throughput for the creation order access pattern noticeably higher than either random pattern. The expected result is for the “random” access pattern to have a lower bandwidth than the “random within a directory” bandwidth that in turn should have a lower bandwidth than the creation order bandwidth due to the increased overhead needed with increasing randomness required to translate the name, find the file’s inode, and find the file’s data. The reason this pattern is not seen here is again attributable to the fragmentation of the filesystem. The expected pattern is seen in the results from running fsbench under FFS in Chapter 5.

While the previous two spatial locality benchmarks are similar to Bonnie, this benchmark is similar to IOStone, except that many of the problems in IOStone are fixed. This benchmark is scalable and has a more realistic filesystem hierarchy. It also separates reading versus writing data, and quantifies what portion of the time is due to meta-data versus data access by measuring the open and close time separately.

However, this benchmark is still not perfect, lacking, for example, a more complete set of access patterns, both inter-file and intra-file.

4.1.5 Phase III: Meta-data Times

This benchmark is the last benchmark in phase III, designed to measure common meta-data operations (create (`open`), delete (`unlink`), `mkdir`, and `rmdir`). A user can use these results combined with the results of the workload characterizer described in Section 4.2 to predict the performance of the workload on the filesystem, while a system designer can use these numbers as rough pointers to possible areas for improvement. These pointers are as accurate as possible, given that a benchmark running at user level can only go so far in pin-pointing problems in the kernel.

The measurements made in this benchmark are the times needed to create *n* files, stat *n* files in creation and random orders, delete *n* files in creation and random order, make *n* directories, stat *n* directories in creation and random orders, and remove *n* directories in creation and random orders. All the measurements are repeated with a `sync()` following each operation. The time needed to complete *n* `sync`’s by themselves is also measured, to determine what the system call overhead is. The value used for *n* in this benchmark is the size of the attribute cache multiplied by the number of repetitions (user-specifiable, see Appendix B): the larger the number of files, the more accurate the final result.

What results are expected from this benchmark depend entirely on how the filesystem is implemented. A filesystem using large meta-data structures and synchronous meta-data writes should perform poorly in comparison to a filesystem using small meta-data structures and asynchronous meta-data operations. Ideally, this benchmark should be coupled with information about the semantic guarantees of the filesystem. For example, under FFS, when a `create` call returns to the user process the user is guaranteed that the file exists on disk. LFS does not have this guarantee (see Chapter 1, Chapter 5). Without this additional information, a user may unknowingly get a win in performance with a potential loss in reliability.

The results from running this benchmark are presented in Table 4.6. By comparing the results with and without a `sync`, it can be deduced that these meta-data operations involve synchronous I/O requests. These results also show the substantial overhead required to deal with directories rather than files, such as allocating blocks for the directory file, writing to disk, and checking that the directory is empty. The difference between sequential and random operations reflects the additional time needed for name translation: finding the inodes, searching the directory file, *etc*. From these results, one possible improvement is to find a way to reduce the overhead in directory operations. It would be up to the designer to decide whether such an optimization should be done by optimizing the code or changing the fundamental meta-data structures used.

Phase	Name	Total Time Elapsed (Std Dev.)	Throughput
III	Create	275205 msec (5801.0)	53 creates / sec
	Stat files (Sequential)	9689 msec (87.0)	1511 stats / sec
	Stat files (Random)	18724 msec (286.7)	782 stats / sec
	Delete (Sequential)	275413 msec (1303.2)	53 deletes / sec
	Delete (Random)	454679 msec (2554.6)	32 deletes / sec
	Mkdir	948724 msec (353.8)	15 mkdirs / sec
	Stat dirs (Sequential)	13345 msec (49.2)	1097 stats / sec
	Stat dirs (Random)	23522 msec (984.6)	622 stats / sec
	Rmdir (Sequential)	758842 msec (592.8)	19 rmdirs / sec
	Rmdir (Random)	1076900 msec (1583.4)	14 rmdirs / sec
IV	Sync	15387 msec (8.1)	951 syncs / sec
	Create (Sync)	277965 msec (636.7)	53 creates / sec
	Delete (Sequential, Sync)	275912 msec (811.8)	53 deletes / sec
	Delete (Random, Sync)	455776 msec (1268.7)	32 deletes / sec
	Mkdir (Sync)	949770 msec (443.7)	15 mkdirs / sec
	Rmdir (Sequential, Sync)	761190 msec (592.8)	19 mkdirs / sec
	Rmdir (Random, Sync)	1081033 msec (1583.4)	14 mkdirs / sec

Table 4.6 **Results from the Meta-data Time Benchmark, using 14,640 files or directories**

4.1.6 Phase IV: Meta-data, Part 2

The first four benchmarks in phase IV are designed to help system designers pinpoint possible problems in meta-data operations more precisely than in the previous meta-data benchmark (Section 4.1.5). Specifically, they try to isolate attribute (inode) create time, directory create time, attribute access time, and name lookup time by timing different meta-data operations and subtracting out the appropriate overlaps.

The first benchmark determines how long it takes to allocate physical space for a new inode (or whatever the corresponding control structure is). To do this, three measurements are taken: time needed to create *n* new files, time needed to stat the *n* files, and time needed to create *n* hard links to the files (1 link per file), where the file is `stat`'d before the link is created. Before each measurement, all the caches are flushed, and the base hierarchy is read in (via `stat`()) to reach a consistent start state. It was initially thought that the time desired was just *create time - hard link time* (without the initial `stat`()), since both creating a file and creating a hard link require an inode, a directory inode, and a directory file to be written, with the create having the extra time needed to allocate the inode. However, the synchronous meta-data read of the file’s attributes made link time higher than create time. This synchronous read needed to be factored out, and therefore the time needed to allocate space for a new inode equals *create time - stat time - link time*.

Similarly, the time needed to allocate space for a new directory equals the time needed to make a directory minus the time needed to create a file (no extra `stat` is needed, unlike for inode create time). As before, all the caches are flushed and the base hierarchy is read in before each of the two measurements. In both cases, *n* equals the inode cache size times the number of repetitions (user specifiable, see Appendix B). Again, this value is used with the underlying idea that the larger the number of files or directories used, the more accurate the final result.

The next two benchmarks, inode access time and name lookup time, are interdependent and difficult, if not impossible, to isolate from one another. The problem is that the system call used, `stat`, involves both a pathname translation and an attribute access. This problem also occurs in trying to size the attribute cache and the name translation cache (see Section 4.1.2). By putting either the needed attributes in the attribute cache, or the needed translations in the name translation cache, and subtracting out the extra cache access time, a lower bound on either the name lookup or the inode access time, respectively, can be found. The implementation used in this benchmark has the attribute access time dependent on the name translation time.

Name lookup has two aspects, translating an entire pathname and finding a file within a directory, and the benchmark uses a different hierarchy for each aspect: name lookup within a directory uses a hierarchy with *n* distinct files, and the pathname translation uses a hierarchy with *n* distinct paths (see Appendix C for the algorithms used to create the different hierarchies). Note that some filesystems handle both aspects uniformly. In this benchmark, *n* is the average of the attribute and name translation cache sizes, so that the attributes will fit in the attribute cache, but all the translations needed may not necessarily fit in the name lookup cache (they will for name lookup within a directory, but not for pathname translation). The assumption made here is that the attribute cache is larger than the name cache.

The hierarchy with *n* files actually consists of two distinct but isomorphic hierarchies, one with files and one with links to the files in the first hierarchy. On this basis, three measurements are made: reading (`stat()`) the links in creation (and alphabetical) order, reading the links randomly within a directory, and reading the links randomly. Before each measurement the name cache is flushed and the file attributes are read into the inode cache using `stat()`. Because the number of files used is smaller than the size of the attribute cache, when the hard links are read, the attributes are in the cache, but the pathname translation is not in the name cache until all of the directories have been read. Because all of the paths will fit in the name cache, this part of the benchmark measures the time needed to find a specific name in a directory file that is in the buffer cache for different access patterns.

The other hierarchy, with *n* distinct directories, has one hard link in each directory to a file in the previous hierarchy. On this hierarchy, the time to read a random permutation of the hard links is measured. As before, the name cache is flushed and the files are read into the attribute cache before each measurement. Because in this case the directory file may not be in the cache, this benchmark measures the time needed to read multiple directory files, perhaps some from disk, and find one name in each.

The inode access benchmark uses the results from the first hierarchy in the name lookup benchmark. In the inode access benchmark, the times needed to read the attributes of the files in the hierarchy with n distinct files both in creation and random order is measured. Before each measurement, all the caches are flushed. The times measured to lookup names in creation and random order, respectively, are subtracted to obtain a lower bound on inode access time.

The results from running all four benchmarks on *sake* are presented in Table 4.7. As can be seen, the time needed to create a hard link is still higher than the time to create a file, even after accounting for the time needed to read the inode. This result suggests one of two possibilities: either the benchmark is not accounting for some operation needed to create a hard link, or creating hard links is an area for improvement.

Another area for improvement indicated by these results is directory create time, which we expected from the previous meta-data benchmark. The operations needed to create a file (also needed when creating a directory) take up only 29% of the time – the rest of the time is spent in directory specific operations, such as allocating space for the directory file and writing the directory file.

One interesting result is that name lookup actually accounts for a higher percentage of the time when accessing inodes sequentially rather than randomly. One might expect that because name translation under a random access pattern takes longer than under a sequential access pattern, it would correspondingly account for a higher percentage of inode access time. However, the additional seeks needed to locate inodes on disk under a random access pattern dominate.

Finally, looking up *n* directories takes much more time than looking up *n* files; this is unsurprising since the translations for the parent directories of the hierarchy with *n* files will fit in the cache, unlike the hierarchy with *n* directories.

Benchmark	Name	Total Time (Std Dev.)	Throughput	Derived Results
InodeCreate	Create	268773 msec (3441.0)	54 creates / sec	Creating a hard link requires more time than creating a file
	Stat	9883 msec (37.3)	1481 stats / sec	
	Hard link	465865 msec (879.9)	31 hardlinks / sec	
DirCreate	Create	275208 msec (43.3)	53 creates / sec	Making a directory takes 3.4 times more time than creating a directory.
	Mkdir	945523 msec (621.4)	15 mkdirs / sec	
Name Lookup	Sequential	692 msec (5.9)	2652 lookups / sec	Name lookup accounts for 16% of the time required for accessing inodes sequentially, and 10% of the time for accessing inodes randomly.
	Random (dir)	780 msec (11.6)	2353 lookups / sec	
	Random (level)	879 msec (6.1)	2088 lookups / sec	
	Random	89899 msec (310.4)	20 lookups / sec	
Inode Access	Sequential	4960 msec (47.3)	1472 stats / sec	See above – Name Lookup.
	Random	9065 msec (88.3)	1089 stats / sec	

Table 4.7 **Results from the second meta-data benchmark, using 14,640 files or directories for InodeCreate and DirCreate, and 1,835 files or directories for the NameLookup and InodeAccess benchmarks.**

4.1.7 Phase IV: Readahead

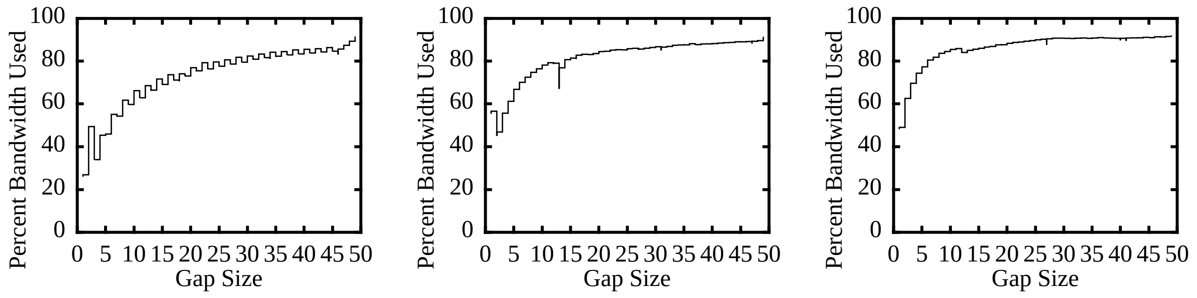
The first data benchmark in phase IV looks for degenerate cases where filesystem readahead detracts from, rather than improves, performance. For example, Seltzer and Smith were running a benchmark that reads random 8 KByte blocks from a file [1]. The filesystem under test was performing horribly, because its block size was 4 KBytes. As a result, the first 4 KByte read would not invoke readahead because of the overall random pattern, but the second 4 KByte read would, thus slowing down the next randomly placed 4 KByte block.

This benchmark times 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,...; ...; 1, 2, 3, 4,... Three different block sizes are used: half the filesystem block size, the filesystem block size, and two times the filesystem block size. The size of the file used equals the size of the buffer cache times the number of repetitions (user specifiable, see Appendix B). The buffer cache is flushed before each test pattern for a consistent start state.

The expected result is for the throughput to increase as the gap size decreases regardless of block size. As the gap size decreases, the number of blocks benefitting from readahead increase and the cost of reading ahead an unnecessary block is amortized.

Figure 4.1 shows the results from running this benchmark on *sake*. Unsurprisingly, the shape of all three curves is the same and is as expected. The one interesting result are the oscillations in the 4 KByte block case; these oscillations depend on whether readahead is invoked on the last block: When the last block read is the first 4 KBytes in an 8 KByte block, then the bandwidth is higher than if it were the second 4 KBytes, because the next (unneeded) block is not read.

Figure 4.1 Readahead Bandwidths for Different Block and Gap Sizes



4 KByte blocks **8 KByte blocks** **16 KByte blocks**

4.1.8 Phase IV: Concurrency

The last micro-benchmark tests how well the filesystem handles concurrent processes making requests, by using the seven benchmarks described in Sections 4.1.3, 4.1.4, and 4.1.6. Because these benchmarks have already been run in isolation, those results are a basis for comparison to the results when more than one benchmark is running concurrently.

The benchmark starts with just two children, and runs all twenty-eight combinations of the seven benchmarks. Then, for three children up to the maximum number of children (user specifiable, defaults to five), *n* different combinations of the seven benchmarks are run, where *n* is user specifiable and defaults to 10.

The results from running this benchmark with two and three children are presented in Table 4.8 and Table 4.9. The two large-file data benchmarks (spatial locality with sequential or random access patterns) both lose when run against any other benchmarks. The sequential access benchmark loses the most bandwidth when reading, because rather than just being able to readahead blocks, now there are seeks to other files or directories mixed in. The meta-data benchmarks lose next to nothing when run with the large-file data benchmarks for three reasons. First, all the caches are flushed in the meta-data benchmarks rather than just the buffer cache; as a result, it is very likely that the part of the data benchmark that is timed for a result is being run while the other benchmark is interfering by flushing the cache. Note that flushing the buffer cache in the data benchmark does not affect the meta-data benchmark very much, while flushing the buffer cache in a meta-data benchmark does affect a data benchmark. Secondly, especially in the sequential access benchmark, read throughput is drastically lower because of the many more seeks mixed in due to the other disk activity. Finally, process management now becomes an issue. Before, when there was only one process, each benchmark could run with no interruptions. Now, with two processes, one benchmark would block to let the other benchmark run, and especially the reads, as mentioned before, would get queued behind other disk accesses and get slowed down. Again, this does not affect the benchmarks with more than one file as much for two reasons: first, the data benchmark is likely to be running concurrently with the cache flushing in the other benchmark (thus affecting the result of the data benchmark and not the meta-data benchmark), and secondly, with the spreading out of files among different directories (see Appendix C), there is much more seeking to different parts of the disk already going on, leading to the original lower throughput. This seems to hold for the three children case as well. Perhaps a better way to test concurrency would be to compare the times needed to run the entire benchmark (by itself in comparison to with other benchmarks). However, due to the death of *sake*, this was not able to be tested.

	SpaLoc - S	**SpaLoc – R**	**LogLoc**	**InodeCreate**	**DirCreate**	**NameL**	**InodeAccess**
SpaLoc – S	75%, 66%	67%, 61%	62%, 99%	76%, 99%	76%, 99%	77%, 99%	67%, 98%
SpaLoc – R		69%, 56%	65%, 99%	81%, 99%	83%, 99%	67%, 99%	73%, 98%
LogLoc			40%, 40%	54%, 46%	77%, 45%	51%, 35%	50%, 73%
InodeCreate				57%, 58%	47%, 32%	52%, 52%	56%, 85%
DirCreate					42%, 58%	60%, 63%	62%, 73%
NameL						51%, 51%	44%, 76%
InodeAccess							66%, 66%

Table 4.8 **Concurrency Results with Two Children.** Both numbers are the average of the percentage of the result from running the benchmark by itself (see previous sections). The first number is for the benchmark in the row, and the second is for the column. Abbreviations used are: SpaLoc – S = Spatial Locality, Sequential Access; SpaLoc – R = Spatial Locality, Random Access; LogLoc = Logical Locality; NameL = Name Lookup.

Child 1	Child 2	Child 3	Percentages
DirCreate	InodeCreate	NameL	26%, 36%, 41%
SpaLoc – S	LogLoc	NameL	63%, 51%, 35%
DirCreate	InodeAccess	InodeAccess	53%, 36%, 58%
SpaLoc – S	InodeCreate	InodeAccess	64%, 40%, 73%
SpaLoc – S	InodeCreate	LogLoc	62%, 46%, 54%
SpaLoc – R	DirCreate	InodeCreate	76%, 20%, 46%
InodeCreate	InodeCreate	InodeAccess	30%, 29%, 58%
InodeCreate	LogLoc	NameL	33%, 41%, 26%
SpaLoc – R	DirCreate	InodeAccess	71%, 44%, 71%
SpaLoc – R	SpaLoc – S	LogLoc	65%, 49%, 83%

Table 4.9 **Concurrency Results with Three Children. Same abbreviations used as above.**

4.1.9 Summary

These micro-benchmarks test several aspects of filesystem design: block allocation to files, readahead policy, meta-data operations, name lookup, and concurrent accesses to the filesystem. As already stated, this is only a first cut at a complete suite of benchmarks to test filesystem functionality and design as laid out in Chapter 3. In Chapter 6, possible extensions to fsbench are presented.

Fsbench tries to adhere to the guidelines of this benchmarking approach stated at the beginning of this chapter. It does meet many of the criteria specified in Chapter 3: it is scalable, it measures much filesystem functionality, it is tightly specified, it is comparable between filesystems running under the same operating system and on the same hardware, and it is prescriptive. It also targets both system designers and users trying to decide which system to buy by dividing the micro-benchmarks into two phases, thus actively acknowledging the split audience and facilitating usability. However, fsbench is not perfect. Before describing the workload characterizer, which combined with the micro-benchmarks yields a generally applicable benchmark, some problems with the implementation of fsbench are discussed.

One problem with fsbench is that it is too UNIX-oriented: Fsbench assumes that filesystems have a hierarchical directory structure, support hard links, and have buffer, attribute, and name translation caches. These assumptions do not hold for all filesystems.

A logistical problem with fsbench is the time needed to run even this rudimentary set of benchmarks. On *sake*, which has a fairly fast disk given today’s technology, this benchmark took approximately one day to run with the number of repetitions set at 5. This is 4 times as long as the time needed to run Chen’s Self-Scaling I/O Benchmark. The large amount of time required to run this benchmark is mainly due to the time needed to flush the caches and the meta-data benchmarks, since, under FFS, meta-data operations are synchronous.

This problem is not as severe as it appears because only if the numbers were being published would the complete set of benchmarks need to be run. In such a situation, a researcher probably has a dedicated test system on which to conduct experiments anyway. Otherwise, for tweaking performance, the appropriate micro-benchmark can be run, which takes much less time. However, to make sure that the tweak does not adversely affect other aspects of the system, the entire suite should be run.

Another problem with fsbench occurs when it is run on a new filesystem. On a clean filesystem, placement of files, directories, and attributes is more optimal than would occur on a fragmented filesystem that would actually exist [1]. The solution to this problem is to age the filesystem. A simple method is presented in Appendix E, and Keith Smith at Harvard is working on a more accurate method to age filesystems.

4.2 Workload Characterizer

The second part of this benchmark is a workload characterizer, used to generate statistics about the target workload. These results can be combined with the statistics gathered about the filesystem using fsbench. Currently, the workload characterizer consists of a perl script that requires as input an NFS trace generated by a modified version of `nfswatch`. Other trace formats can be accommodated by either writing a wrapper script to transform the input trace into the format required or modifying the current scripts to handle the input trace format. A workload generator for the workload characterizer would be a useful tool, since many users do not have a trace of their workload, or may want to analyze the performance for a projected workload. In Chapter 6, previous workload generators and a new idea for a workload generator is discussed.

The statistics gathered by the workload characterizer script are:

- Percentage of calls that are: reads, writes, creates, deletes, mkdirs, rmdirs, readdirs, getattrs, and lookups.
- Percentage of reads that are sequential and random
- Percentage of writes that are sequential and random
- Average number of open files, and the standard deviation from that average

If a logical snapshot (*e.g.*, the output from running `ls -ilR` on the traced filesystem) were also incorporated, statistics about file reference patterns, such as how many files are referenced in the same directory versus how many files are referenced from across the filesystem hierarchy, could also be gathered.

The purpose of the workload characterizer is to make this benchmarking approach generally applicable. The suite of micro-benchmarks alone do not suffice, since by themselves, they cannot be used to determine the performance of a system under a real workload. However, by combining the results from a suite of micro-benchmarks such as fsbench with the statistics generated by a workload characterizer, such as that described above, the two components together can form a complete filesystem benchmark, which can not only help system designers find possible areas for improvement in the system, but also predict the performance of the filesystem under any workload and thus be useful for users trying to decide which system to run.

The difficult part of the above goal is discovering a way to combine the two sets of results to yield an accurate prediction for the performance of the filesystem under the workload.

The idea presented in this thesis is to use a calculation similar to that used to determine the average number of cycles per instruction (CPI) for a processor is used. In a CPI calculation, the number of cycles required for an instruction is multiplied by how often (percentage) that instruction occurs, and the sum over the different instructions is equal to the average number of cycles per instruction. Similarly, to predict how well a workload will perform on a filesystem, the following calculations are made:

- For each meta-data operation: *how often that operation occurs in the workload (percentage) is multiplied by the throughput measured in the first meta-data benchmark*. Readdirs are multiplied by throughput for sequentially stat'ing files, deletes and rmdirs by the throughputs from the random pattern, lookups by the throughput for randomly `stat`'ing directories, and getattrs by the throughput for randomly `stat`'ing files averaged with the throughput for randomly `stat`'ing directories (with a logical snapshot, the number of getattrs on directories and the number of getattrs on files can explicitly determined).
- For data operations: *The percentage of reads * (the percentage of sequential reads * the percentage of the disk bandwidth attained on a sequential read of a file + the percentage of random reads * the percentage of the disk bandwidth attained on a random read of a file)*, and similarly for writes
- For data operations: *The average number of open files * (the percentage of reads * the percentage of the disk bandwidth attained on reading files in random order + the percentage of writes * the percentage of the disk bandwidth attained on writing files in random order)*. Again, with a logical snapshot, the percentage of reads and writes that fall within the different access patterns can be explicitly determined.

In general, with a logical snapshot, different access patterns to files (rather than within a file) can be determined. Note that incorporating a logical snapshot has not been implemented due to time constraints.

This characterization maps the performance of different aspects of the filesystem to their usage in the target workload. In this way, a system with bad meta-data performance might still perform well under a database workload where all the databases are single, large files. Also, the reason to use the percentage of the disk bandwidth attained rather than the actual throughput for the data reads and writes is to factor out the underlying hardware.

4.3 Summary

In this chapter, a filesystem benchmarking approach was presented, consisting of two separate components: a suite of micro-benchmarks and a workload characterizer. Together with the approach, an actual implementation, fsbench plus a perl script, was described.

Recall from Chapter 3 the criteria used to judge a filesystem benchmark: prescriptive, filesystem-bound, scalable, comparable, generally applicable, and tightly-specified. This benchmark fulfills these criteria. It scales by measuring cache sizes (even if it does take forever to run), it is generally applicable with the separation of the workload from the filesystem, and the results it presents are comparable and prescriptive. It is also tightly specified, with the running conditions and report format stated. The main failing of this benchmark is that it is not complete; fsbench needs to be extended to measure cache behavior, for example, and the workload characterizer needs to take into account different access patterns, both for references within a single file and references to different files.

4.4 References

[1] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, V. Padmanabhan, “File System Logging versus Clustering: A Performance Evaluation”, Proceeding of the 1995 USENIX Technical Conference, 249-264.

CHAPTER 5 Using the Benchmark

This chapter presents an example of how to use the benchmark presented in the previous chapter. There are four parts to this experiment: running fsbench on two different filesystems (FFS and LFS), characterizing a target workload, predicting which filesystem would be better for that workload, and then testing that prediction.

The system used for this experiment is *virtual8*, a 20 MHz Sparcstation I running an experimental 4.4 BSD-Lite kernel with LFS fixes (ordered blocks and vnodes, fragments) and extensions to support a journaling filesystem. The disk used is a Fujitsu M2694EXA, a 1 GByte, 5400 rpm disk with a 9.5 ms average seek time and 512 KByte on-disk cache. Note that both FFS and LFS use the same underlying operating system and hardware. With the same base configuration, any differences between the performance of FFS and LFS is due solely to the design and implementation of the filesystems. If the same base configuration were not used, then the comparisons would be questionable, since performance differences might be attributable to other aspects of the operating system.

5.1 Fsbench results from FFS and LFS

Table 5.1 presents the results from running the initial disk measurements on the Fujitsu disk of *virtual8*. Note that the write bandwidth to the block device is very low; even if every write is synchronous and requires two rotations (one to rotate to the appropriate position, and one to write the 8 KByte block), the expected bandwidth is still 360 KBytes / sec – approximately twice what is actually seen. The only reason that might explain this behavior is that perhaps each write is a `write and verify` rather than just a `write`.

Table 5.2 presents the results from running fsbench on a filesystem running FFS and LFS on the same disk (sd1). Note that FFS has 8 KByte blocks and 1 KByte fragments while LFS has 4 KByte blocks and no fragments at all. The filesystem under test was aged using the algorithm outlined in Appendix E.

Name	Read Measurement	Write Measurement
rawDiskBandwidth	1304 KBytes / sec	1264 KBytes / sec
diskBandwidth	1089 KBytes / sec	172 KBytes / sec
rawDiskSeekTime	13 msec	14 msec
diskSeekTime	14 msec	20 msec

Table 5.1 **Initial Disk Measurement Results**

Benchmark	Measurement	Result under FFS	Result under LFS
Spatial Locality, Sequential Access	Write & Allocate	729 KBytes / sec (424%)	1266 KBytes / sec (736%)
	Read	1550 KBytes / sec (142%)	1328 KBytes / sec (122%)
	Overwrite	748 KBytes / sec (435%)	1202 KBytes / sec (699%)
Spatial Locality, Random Access	Write & Allocate	652 KBytes / sec (379%)	1201 KBytes / sec (699%)
	Read	492 KBytes / sec (41%)	315 KBytes / sec (29%)
	Overwrite	766 KBytes / sec (445%)	1105 KBytes / sec (642%)
Logical Locality, Creation Order	Read	1237 KBytes / sec (114%)	1136 KBytes / sec (104%)
	Write	544 Kbytes / sec (316%)	898 KBytes / sec (522%)
Logical Locality, Random W/in Dir.	Read	1138 KBytes / sec (104%)	1059 KBytes / sec (97%)
	Write	534 KBytes / sec (310%)	839 KBytes / sec (488%)
Logical Locality, Random	Read	1091 KBytes / sec (100%)	1045 KBytes / sec (96%)
	Write	527 KBytes / sec (306%)	848 KBytes / sec (493%)
Metadata Time	Create	39 creates / sec	354 creates / sec
	Stat, Sequential	654 stats / sec	608 stats / sec
	Stat, Random	275 stats / sec	260 stats / sec
	Delete, Sequential	83 deletes / sec	636 deletes / sec
	Delete, Random	34 deletes / sec	461 deletes / sec
	Mkdir	7 mkdirs / sec	160 mkdirs / sec
	Stat, Sequential	524 stats / sec	491 stats / sec
	Stat, Random	80 stats / sec	250 stats / sec
	Rmdir, Sequential	10 stats / sec	175 rmdirs / sec
	Rmdir, Random	10 stats / sec	65 rmdirs / sec

Table 5.2 **Results from fsbench**

From the FFS results, the following observations can be made:

- Sequential read bandwidths are high due to an aggressive readahead policy (*e.g.*, setting filesystem parameters so as to optimize reads. On *virtual8*, rotational delay is set to 4 and maxcontig to 7).
- In the logical locality benchmark, note that the original expectation that bandwidths would decrease with increasing randomness is met here. Also, remember that because asynchronous writes in this benchmark are mixed in with synchronous meta-data reads and because of the aggressive readahead, the write bandwidths are substantially lower than the read bandwidths.
- Directory operations take enough time that name lookup time is minimal when removing directories, thus the access pattern used (sequential versus random) does not significantly impact throughput. This is not surprising, given the FFS layout policy of putting different directories in different cylinder groups and putting files in the same directory in the same cylinder group.

In comparison, the observations from the LFS results are:

- Writes are asynchronous in both filesystems, thus the better throughput numbers under LFS are due to two other reasons. First, the largest file used is 8 MBytes; when even smaller files are used in the logical locality benchmark, meta-data throughput impacts performance even more. The increase in overall throughput under LFS corresponds both to an increase in meta-data write throughput and data write throughput (sequential under LFS, not necessarily so under FFS) in the logical locality benchmark. More importantly, the file hierarchies used spread files among different directories, which hurts FFS performance but helps LFS performance, given the differences in disk layout algorithms: FFS aims for future logical locality, while LFS exploits current temporal locality.
- However, while write performance improves, read sequential performance decreases mainly because clustered reads were turned off due to an oversight that was noticed much too late.
- Random read performance decreases significantly as well, due to the different block sizes used. FFS used an 8 KByte block, whereas LFS used a 4 KByte block. As a result, reading the file randomly required more seeks under LFS than under FFS.
- In the logical locality benchmark, the two random write bandwidths were approximately the same, reflecting the fact that LFS does not need to find and overwrite previously allocated blocks.
- The meta-data write numbers were much higher, reflecting the asynchronicity of the operations. As a result, the impact of name lookup can be seen in the throughput for removing directories in random order.
- Like the data read performance, the meta-data read performance, in general, also decreased. This is partially due to turning off clustered reads. The one exception to this degradation in performance is in stat'ing directories in random order, which has performance similar to stat'ing files in random order. The reason for this exception lies in the layout policy of LFS, which ignores logical locality and aims for temporal locality. As a result, because the directories were all created together, all the directories and their inodes are placed close together on disk, leading to fewer seeks and thus yielding better performance. Note that the performance for stat'ing directories in sequential order is worse than that for stat'ing files sequentially. This is due both to turning off clustered reads and to interleaving inodes and directory files on disk.

5.2 Characterizing a Workload

The workload used in this experiment is actually a NFS trace of the filesystem traffic gathered from the system group at Harvard over the course of ten days in October, 1994. This trace consists of accesses to only one server, *attic*, which is a dedicated NFS server made by Network Appliances. A modified version of `nfswatch`, a utility that watches the ethernet for NFS requests, was used to gather the trace. Normally, it just increments counters to gather statistics; it was modified to log all NFS requests to *attic*. These traces were originally used in a paper by Blackwell, et al. to find better heuristics to decide when to invoke the cleaner for LFS [2]. Note that the original trace was based on over 4 GBytes of data and over 5 million requests. Because the disk on *virtual8* is only 1 GByte in size, a data set approximately one-tenth the size of the original was randomly chosen from the original data-set, and the trace was pared appropriately.

The results from running the scripts on the trace, checkpointing every 100,000 requests, are in Table 5.3. Given the origin of the trace (NFS), it is not surprising that over 50% of the requests are meta-data requests, and that the majority of those meta-data requests are read requests. What is more surprising is the ratio of reads to writes and the ratio of sequential and random writes in comparison to earlier studies [1]. The read to write ratio is understandable given that the trace is a trace of requests to a NFS server; as a result, many of the read requests that are fulfilled by the client caches are not captured. The sequential and random ratio for writes in the first 100,000 requests could just be a remnant of being at the beginning of the trace and not capturing earlier requests; this conjecture is borne out by the decrease in the percentage of random writes throughout the rest of the trace. Even so, the number of random writes is still higher than expected; this may be an effect resulting from the randomly chosen data set used.

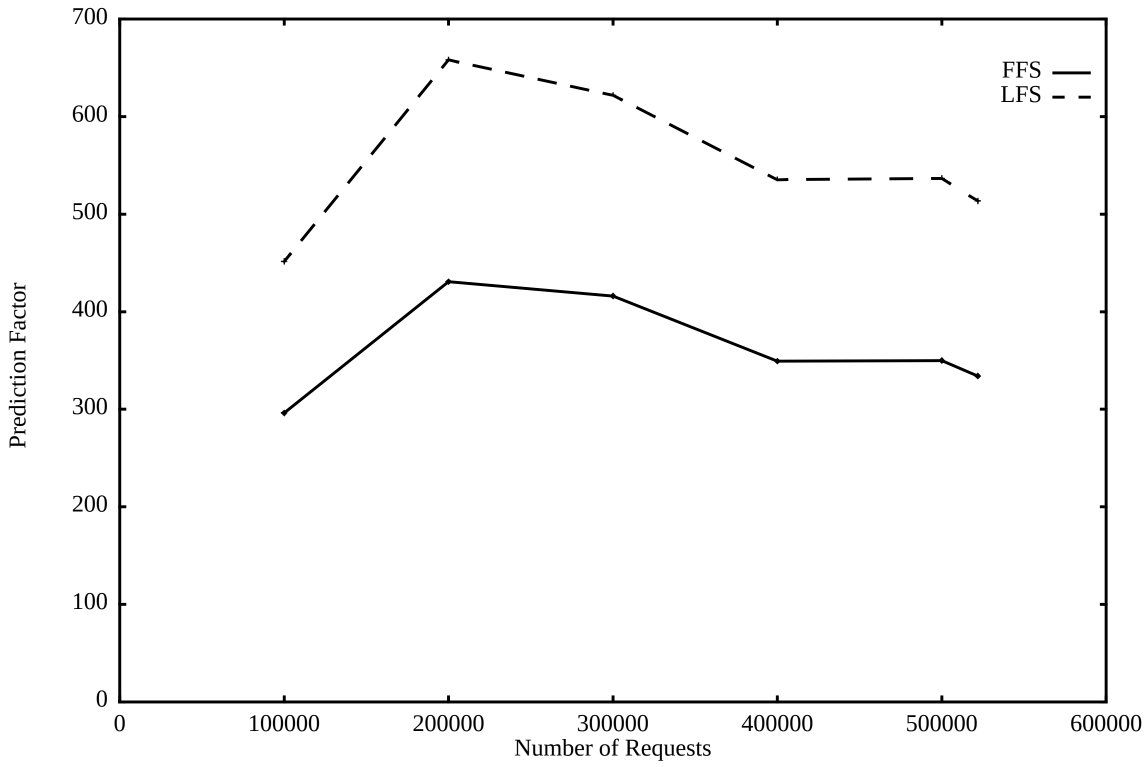
5.3 Predictions

With the statistics in Section 5.1 and 5.2 and the mapping algorithm described in Chapter 4, Figure 1 presents the predictions based on this information. In general, LFS should outperform FFS by a factor of 1.5 to 1.75: *i.e.*, LFS should finish the trace in about 75% of the time it takes FFS to finish running the trace. From the actual shape of the curves, several predictions can be made. First, the FFS and LFS curves when plotting the actual time needed to complete the trace should also be roughly the same shape. Secondly, with the higher performance prediction factors for the second and third 100,000 requests, both filesystems should also finish those requests faster than the last series of requests.

Number of Requests	100,000	200,000	300,000	400,000	500,000	521,883
% Sequential Reads	95%	98%	99%	99%	99%	99%
% Random Reads	5%	2%	1%	1%	1%	1%
% Reads	14%	26%	24%	19%	17%	16%
% Sequential Writes	59%	81%	83%	81%	81%	81%
% Random Writes	41%	19%	17%	19%	19%	19%
% Writes	18%	28%	27%	22%	19%	18%
Avg. # Open Files (Std. Dev)	1.34 (13.5)	1.78 (17.0)	1.78 (19.9)	1.59 (24.1)	1.93 (75.6)	1.87 (72.6)
% Creates	0.4%	0.3%	0.5%	0.4%	0.5%	0.5%
% Deletes	0.1%	0.2%	0.2%	0.2%	0.3%	0.3%
% Mkdirs	0.03%	0.03%	0.03%	0.02%	0.02%	0.02%
% Rmdirs	0.001%	0.001%	0.004	0.01%	0.02%	0.02%
% Getattrs	42%	24%	26%	26%	31%	32%
% Lookups	22%	30%	19%	29%	29%	29%
% Readdirs	2%	1%	2%	3%	3%	3%

Table 5.3 **Workload Characterization**

Figure 5.1 Prediction for FFS and LFS performance



5.4 Testing the Prediction

The predictions made in Section 5.3 were tested by running the trace on both FFS and LFS. The performance metric used is the time needed to complete the operations in the trace.

Running the trace consists of three phases: creating the base hierarchy, aging the hierarchy, and actually completing the operation in the trace on the filesystem. Creating the base hierarchy was difficult, since no logical snapshot of the filesystem was taken before the trace began. As a result, a perl script was written to determine which inode numbers in the trace mapped to directories and which ones mapped to files based on the associated operations. The sizes of files were determined by finding the *maximum offset + requested size* encountered. Given this mapping between inode number and directory or file, a hierarchy was randomly created, using modified versions of the algorithms described in Appendix C: the names were already pre-determined, and rather than putting all the files at the same level in the hierarchy, files were randomly placed into directories. This artificial hierarchy was put onto a new filesystem and aged using the algorithms described in Appendix E. To run the trace on the hierarchy, operations in the trace were mapped to actual filesystem calls: a getattr or a lookup became a `stat`, a setattr became a `chmod`, *etc*. The timestamps on the original trace were ignored because there was no way of determining which requests were dependent on one another and which ones were not. As a result, the assumption made was that all requests were interdependent, and therefore one operation began as soon as the previous one completed.

Figure 5.2 presents the results from running the trace, and Figure 5.3 compares the predicted ratios of performance to the actual ratios. As can be seen, the prediction algorithm is not very good. It does manage to predict that LFS will perform better than FFS; however, it does not predict the actual ratio very well. The problem with the mapping function is that the different operations, meta-data and data, are weighted differently: one uses actual throughput while the other uses percentage throughput attained, and the data reads are weighted in twice (once for within one file performance and once for inter-file performance).

Due to time constraints, this mapping function has not been thoroughly considered nor manipulated. However, we believe that this is the right approach, and with appropriate adjustments to the mapping function, based on the above observations, that the performance of filesystems can be accurately predicted.

Figure 5.2 Performance of FFS and LFS on the trace

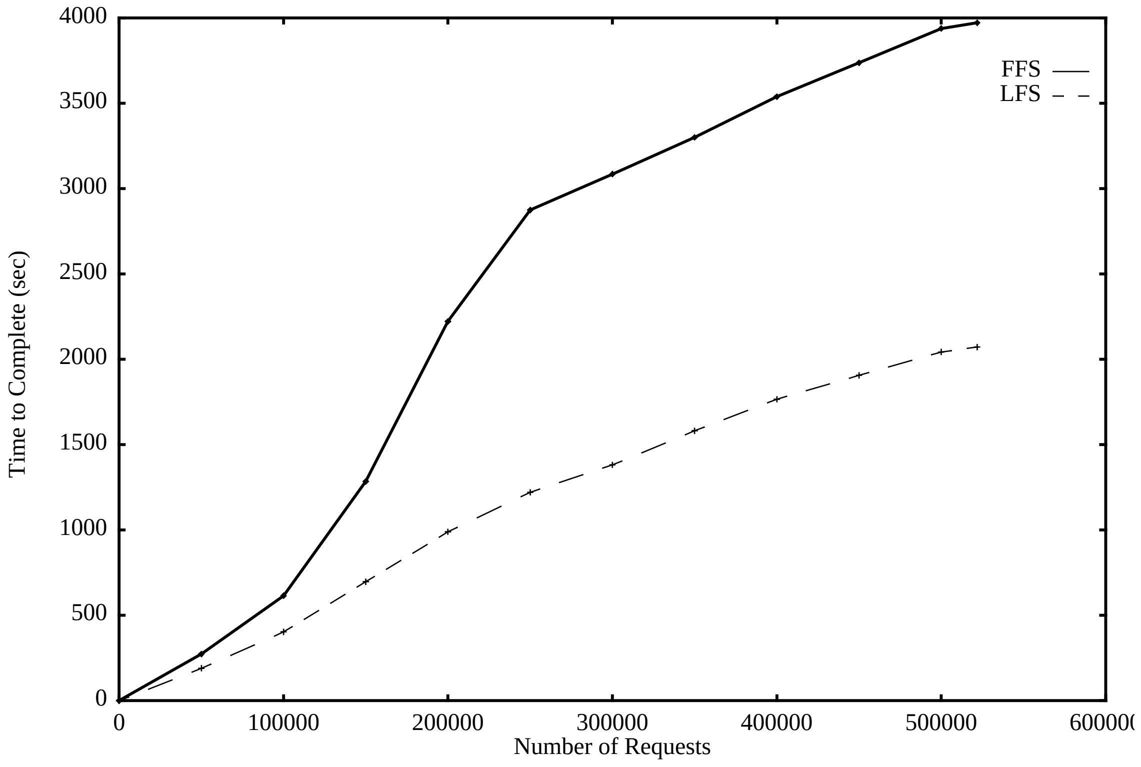
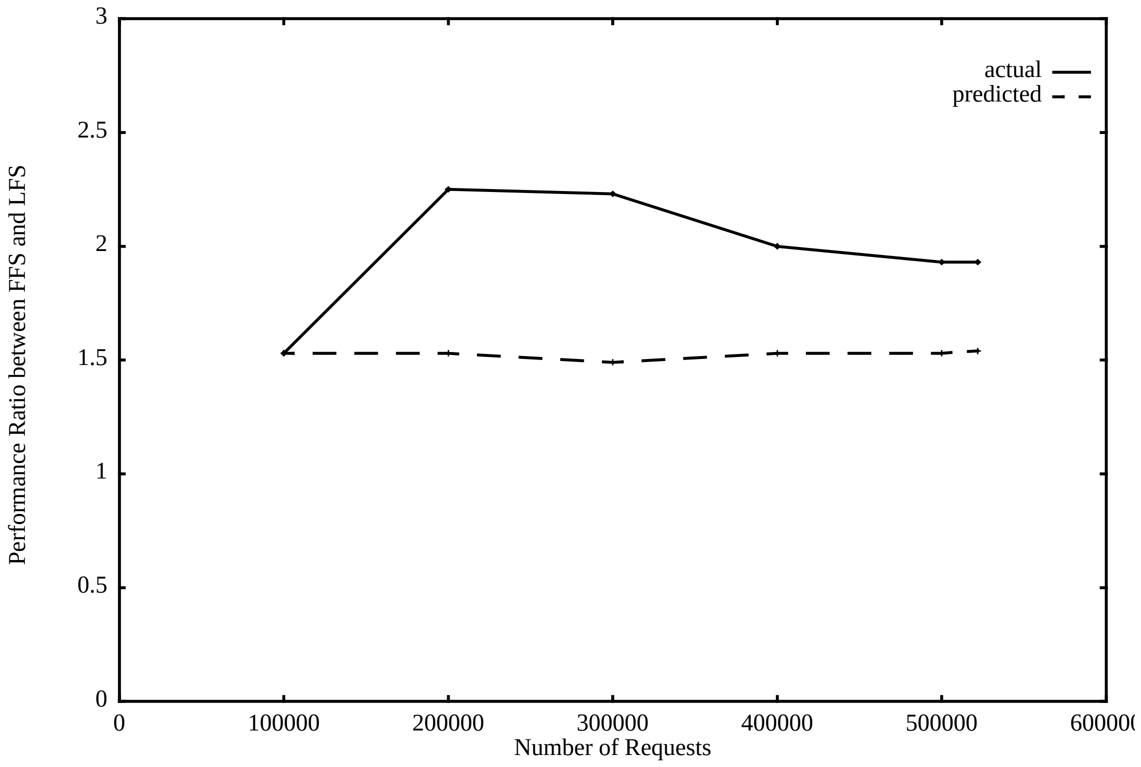


Figure 5.3 Comparison of Predicted Performance to Actual Performance



5.5 References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. “Measurements of a Distributed File System”, Proceedings of the 13th Symposium on Operating Systems, 1991, 198-212.

[2] T. Blackwell, J. Harris, M. Seltzer. “Heuristic Cleaning Algorithms for Log-Structured File Systems”, Proceedings of the 1995 USENIX Technical Conference, 277-288.

CHAPTER 6

Conclusion and Future Work

Existing benchmarks used to measure filesystems are inadequate, lacking full functionality, scalability, and general applicability, and possibly presenting results in a meaningless format. This thesis has tried not only to point out what is wrong with current benchmarks, but also to determine criteria by which to judge benchmarks and the needed functionality in a filesystem benchmark, and to propose and implement a benchmarking methodology that meets these goals.

In this last goal, with dtangbm, this thesis has only partially succeeded. On the one hand, dtangbm is prescriptive, scalable, and tightly specified. It measures much more filesystem functionality than previous benchmarks, although it is not complete (see Section 6.1 for extensions). It is also comparable between different filesystems running under the same operating system, and if extended (see Section 6.3), it can be comparable between systems. Unfortunately, dtangbm is not generally applicable at this moment because its prediction algorithm is not tuned correctly. As a result, this benchmark is still useful for system designers, both to better understand their system and to tune their system, but because the prediction algorithm is not accurate enough, those looking to use this benchmark to decide which system to buy should wait until the next release.

The prediction algorithm is not as essential for system designers anyway; while they would like to compare systems, they can directly compare the results from fsbench to determine which differences between filesystems probably caused which performance results. And while they would like to determine which aspects of the system to change in order to improve performance the most, again, they can just look directly at the results of fsbench and the workload characterizer to get a good idea. It is only if an accurate comparison between systems is needed that the benchmark fails – it can give a rough idea, but not much beyond that. However, this approach should not be ruled out until it is more thoroughly investigated.

Possible directions for future work, besides improving the prediction algorithm, include enhancing fsbench, writing a workload generator, and extending the dtangbm to benchmark the entire operating system.

6.1 Extending fsbench

As stated previously, fsbench is merely a rudimentary suite of filesystem micro-benchmarks. Possible extensions to fsbench include more initial disk measurements to discover the disk's caching algorithm, a temporal locality micro-benchmark, a micro-benchmark to determine filesystem cache management, a disk scheduling (queuing) micro-benchmark, and an extension to fsbench for distributed filesystems.

The purpose of discovering the disk's caching algorithm is to help determine what performance gain is due to a clever filesystem and what is due to a clever disk. Most SCSI disks made today are very complex and have track buffers to cache data for read requests. The Seagate disk on *sake*, for example, can recognize a sequential access pattern and start reading ahead sectors. With a random access pattern, on the other hand, the disk will not bother to expend the additional overhead needed for readahead. Some disks, such as the DEC disk on one of Harvard's Alpha flashovers, put everything that passes underneath the head into the track buffer.

Robert Morris wrote `rtmbench` to try to determine disk caching policies (finding track-to-track seek times as well). `Rtmbench` iterates over different gap sizes. For each gap, it reads a random sector, *n*, from the disk, and then measures the time needed to read sector *n+gap* from the disk to determine whether or not sector *n+gap* was in the track buffer (see Figure 6.1 for an example). Unfortunately, this benchmark by itself is too simplistic to determine the disk's caching policy since different access patterns may evoke different caching policies. Just as importantly, mapping this result to filesystem performance is non-trivial.

Another extension for fsbench is to discover when data is aged out of the cache. A micro-benchmark to complete this task has actually been written, but takes an unreasonable amount of time to complete. The current benchmark makes three measurements using a file the same size as the estimated size of the buffer cache. It measures how long it takes to change (write) one byte per filesystem block, iterating over all the blocks in the file *n* times, where *n* is the number of repetitions (see Appendix A). The time to `fsync` the file when all of its blocks in the buffer cache are clean is the second measurement made; `fsync` returns when all the dirty blocks in the file have actually been written to disk. Again, this measurement is repeated *1000 * n* times to try to increase the accuracy of the final average. Finally the main loop iterates over *i*, and it is this loop that takes the most time. Within this loop, the time needed to change one byte, sleep *i* seconds, and then `fsync` the file is measured and repeated for each block in the file *n* times. If this time is approximately the same as *the time measured to change one byte of the file + the time to `fsync` a clean file + i*, then the data has already been flushed from the cache. If this is the case for all of the blocks in the file, then *i* is an upper bound on approximately how often `update`, the daemon that flushes dirty data in the buffer cache to disk, is run. While this benchmark works, it takes much too long to be practical. This information, combined with information such as the average lifetime of files, can be used to determine if *i* is set correctly.

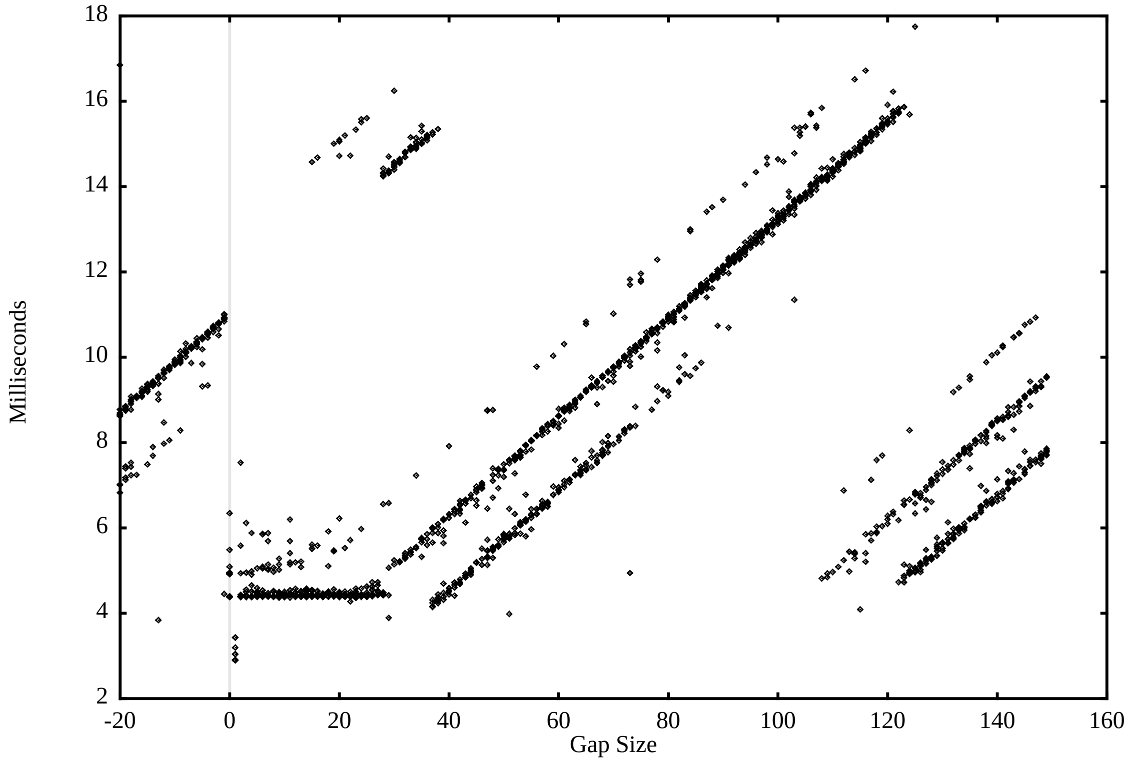
Fsbench should also be extended to include a temporal locality benchmark to match the existing spatial and logical locality benchmarks and a write analog for the readahead microbenchmark. For example, the benchmark could append to a file (write to the file, sleep for five minutes, and write some more to a file), and then read the file. Under a filesystem that ignores temporal locality, like FFS, reading the file sequentially will yield the same result no matter how the file is written. Under a filesystem that exploits temporal locality, like LFS, reading the file sequentially is much more difficult.

The purpose of the write analog to the readahead microbenchmark is to test the filesystem's disk scheduling algorithm and to look for degenerate write patterns. For example, one pattern might be to write fifty blocks, and then overwrite the last block to see whether requests already queued to disk can be dequeued. This particular example would not give the desired answer in a system where writes are asynchronous.

The last extension discussed here is to enhance fsbench to measure distributed filesystems. To do this, a microbenchmark to measure network or RPC latency is needed, as well as some method of determining how often data is lost.

Other extensions might include a latency test, a security benchmark, and a crash recovery benchmark.

Figure 6.1 `Rtmbench` Output. From this graph, it can be determined that 25-30 sectors immediately following the sector read are cached, and that the track-to-track seek time is approximately 3 milliseconds. However, this is only the result for a random pattern. If, for example, two sectors were read at a time, the disk might start reading ahead sectors. SCSI disks are complex, and a more complex benchmark is needed to even try to understand their behavior.



6.2 A Workload Generator

Another area of future for future work would be to devise a workload trace generator to generate input to the workload characterizer, described in Chapter 4.2, since a real input trace can be difficult, if not impossible, to obtain. This generator would take as input a few parameters whose values are easily obtainable by users, such as the size of the filesystem (depth, number of files, number of directories) and the average number of running processes.

Previous work in workload generation includes SynRGen from CMU and a workload model for distributed file servers from the University of Saskatchewan. SynRGen is designed to generate filesystem traces at the system call model; it models a workload by combining micro-models, where a micro-model models the file reference patterns of an application [2]. The two problems with SynRGen are that it does not capture inter-application dependencies, *e.g.*, saving files in an editor before compiling, and that it is difficult to model many different workloads until the underlying micro-models have been written, which is a non-trivial task. The work from Canada generates workload models for distributed file servers, specifically NFS, using four parameters: the frequency distribution of requests to the server (*e.g.*, what percentage of the requests are reads), the request interarrival time distribution, file referencing behavior (which files are touched when), and the distribution of sizes of read and write requests [1]. While this work looks promising, it is currently only useful for NFS workload generation, rather than being more generally applicable.

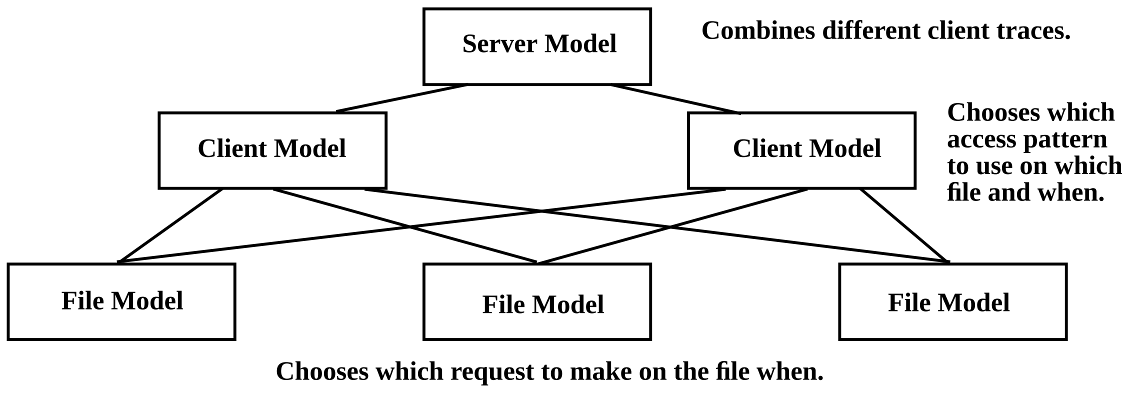
There has been much work in trying to attain the holy grail of workload generation. Another idea, somewhat similar to both projects mentioned above, is presented here. The idea is to use a hierarchical statistical generator. The lowest level of the hierarchy would consist of a suite of models, each one modeling a different file access pattern. For example, the pattern for a system binary is probably create, write, close, and then open, read, close repeated over and over again; a temporary file's access pattern is probably create, write, and then delete. This level of modeling generates filesystem requests for one file. The next level generates a workload trace by combining different file traces; while the file-level model only had to choose what requests to make when, this level must choose which access pattern to start when and on what file or directory. With this division of labor, the modeling becomes simpler because each model is responsible for fewer parameters. It also becomes more realistic because inter-file dependencies can be incorporated; for example, the make application will open, read, and close many source files in the same directory or source tree before creating, writing, and closing several object files and executables in some target directory. Another level can be added on top of this level to generate a load for a distributed file system; this level would combine the client workloads to generate a server workload. See Figure 6.2 for an overview of the approach.

These top two levels of the hierarchy will be the most difficult to implement in the generator. Because they combine traces, they need to make some assumptions about the client or server. Some requests are inter-dependent, and might require, for example, an assumption about the response time of the machine; independent requests require no such assumption. Determining when requests are dependent on filesystem response time, CPU speed, *etc*., and then determining the appropriate inter-arrival time, is non-trivial.

However, if this idea is successful, then it has several advantages over the previous methods. Because it is at the filesystem system call level, it can be used to model any filesystem, whether it be FFS, LFS, NFS, AFS,... It is also scalable in the load on a client, the number of clients, and the size of the filesystem (by simply increasing the size of the name space used in the second level of the hierarchy). More importantly, it is fairly representative of reality, modeling not only inter-file dependencies within an application, but also inter-application dependences. For example, a user might be editing files in one window, and save all of those files before going to another window to compile the files.

Unfortunately, this idea has not yet been tested, and therefore this is all conjecture about a possible solution to the holy grail of workload generation.

Figure 6.2 Overview of the Hierarchical Statistical Generator


Server Model
Combines different client traces.
Client Model
Client Model
Chooses which access pattern to use on which file and when.
File Model
File Model
File Model
Chooses which request to make on the file when.

6.3 OSbench

The last path for future work discussed here is using this method to characterize an operating system, rather than just the filesystem. One problem in several research papers is that one aspect of the operating system is compared under two different operating systems. For example, in the original LFS paper, FFS under SunOS and LFS under Sprite – two different operating systems – were compared to show that LFS performed better than FFS in general [3]. In such a situation, the question arises as to whether the performance gain can be attributed to some aspect of Sprite other than the filesystem.

If fsbench were extended to measure the entire operating system, then perhaps such cross-platform comparisons could be made to alleviate such questions. Aspects such as the virtual memory system, which tends to be closely linked to the filesystem, and process management need to be measured, and interdependencies discovered and accounted for.

6.4 References

[1] Bodnarchuk, R. R., Bunt, R. B. “A Synthetic Workload Model for a Distributed System File Server”, Performance Evaluation Review (19, 1), June 1991, p. 50-59.

[2] Ebling, M. R., Satyanarayanan, M. “SynRGen: An Extensible File Reference Generator”, Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May, 1994, p. 108-122.

[3] M. Rosenblum, J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System”, ACM Transactions on Computer Systems, February 1992, 26-52.

Acknowledgments

I'd like to thank:

- Margo Seltzer, my thesis advisor
- Mike Smith and Brad Chen, other members of my thesis committee
- Keith Smith, as one in the know about filesystems
- Robert Morris, a very smart person
- Cliff Young, Trevor Blackwell, LeeAnn Tzeng, and Thomas Charuhas, my friends
- James Gwertzman, a fellow thesis writer
- my parents
- and Erik

Appendix A

`Iostat` *and* `Vmstat`

Table A.1 summarizes `iostat`, and Table A.2 summarizes `vmstat`. For more information, see the *4.4 BSD System Manager's Reference Manual*, published by O'Reilly Associates.

Field	**Description**
tin	number of characters read from terminals in the past <interval> seconds
tty	number of characters written to terminals in the past <interval> seconds
sps (for each disk)	sectors transferred per second (averaged over past <interval> seconds)
tps (for each disk)	transfers per second (averaged over past <interval> seconds)
msps (for each disk)	milliseconds per average seek (including implied seeks and rotational latency)
us	percentage of cpu time in user mode over past <interval> seconds
ni	percentage of cpu time in user mode running niced processes
sy	percentage of cpu time in system mode
id	percentage of cpu time in idle mode

Table A.1 **`iostat`:** Every <interval> seconds, for a total of <count> times, `iostat` outputs a line with these fields in it.

Field	Description
r	number of processes in the run queue
b	number of processes blocked for resources (*e.g.*, i/o, paging)
w	number of processes runnable or short sleeper (< 20 seconds), but swapped out
avm	number of active virtual pages (*i.e.*, belonging to processes that are running or have run in the last 20 seconds)
fre	number of virtual pages on the free list
flt	number of page faults per second, averaged over the past 5 seconds
re	number of page reclaims per second, averaged over the past 5 seconds
at	number of pages attached, averaged over the past 5 seconds
pi	number of pages paged in, averaged over the past 5 seconds
po	number of pages paged out, averaged over the past 5 seconds
fr	number of pages freed per second, averaged over the past 5 seconds
de	number of pages anticipated short term memory shortfall, averaged over the past 5 seconds
sr	number of pages scanned by the clock algorithm, averaged over the past 5 seconds
disk	number of disk operations per second (paging is usually split across available drives)
in	number of device interrupts per <interval> (including clock interrupts), averaged over the past 5 seconds
sy	number of system calls per <interval>, averaged over the past 5 seconds
cs	cpu context switch rate (switches/interval), averaged over the past 5 seconds
us	percentage of CPU time spent in user mode
sy	percentage of CPU time spent in system mode
id	percentage of CPU time spent in idle mode

Table A.2 **`vmstat:`** Every <interval> seconds, for a total of <count> times, `vmstat` outputs a line with these fields in it.

Appendix B

fsbench: Command-line Options

All of the command-line options for running fsbench are described in Table B.1. The one required option is the working directory for the benchmark. If the initial benchmarks are run, then the device name and size need to be specified, and the scripts `mountFS` and `umountFS` need to be customized as well. If the benchmark is allowed to write to the raw disk (`-n` specified), then the `newFS` script needs to be customized as well. The option, `-r`, which specifies the number of repetitions required, is an accuracy option. It is used to determine how many repetitions of each benchmark to run and to determine the size of files and the number of files used. The idea is that the larger the number of repetitions, the longer fsbench takes to run, but the more accurate the final result

Other options currently not implemented but trivial to add are options to allow the user to specify the various cache sizes rather than running phase II and options to specify which subset of micro-benchmarks to run.

Another possible extension would be to separate the number of times each benchmark is repeated from the multiple of the cache sizes used in the various benchmarks. In this way, the cache size multiple option can be used to increase the scalability of the benchmark.

Option	Argument	Default	Description
-w	<dir path>	N/A	Tells the benchmark in which directory to run all of the tests.
-i	N/A	off	Tells the benchmark to run the phase I
-d	<device name>	N/A	Tells the benchmark which device to run the initial disk measurements on (e.g., sd0). This option is required if phase I is being run.
-s	<size>	N/A	Tells the benchmark how large the device specified with -d. This option is required if phase I is being run.
-n	N/A	off	Tells the benchmark that it is allowed to run the write tests in phase I.
-p	<processes>	5	Tells the benchmark the maximum number of child processes to spawn for the concurrency test in phase IV.
-m	<permutations>	5	Tells the benchmark the number of different combinations of micro-benchmarks for the concurrency test in phase IV.
-r	<repetitions>	5	Tells the benchmark how many times to run each benchmark.
-v	N/A	off	Tells the benchmark to run phase IV.

Table B.1 **Command-line Options for fsbench**

Appendix C

fsbench: Creating Hierarchies

Fsbench has two functions that create filesystem hierarchies: one that creates a hierarchy with *n* distinct files, and one that creates a hierarchy with *n* distinct paths, *i.e.*, *n* distinct directories, where *n* is a parameter passed into the function. These functions were created so that all of the files and directories would not be placed in the same directory, because if *n* is large, then name lookup time within the directory becomes disproportionately large.

For the hierarchy with *n* distinct files, all *n* files are placed at the same level in the hierarchy (also the lowest level in the hierarchy). The rule used is that no more than one hundred files or directories are allowed to be in a directory. Thus, the number of levels in the hierarchy equals the ceiling($\log_{100}$ n), where *n* is the total number of files. This value could be easily changed to be user specifiable.

Creating the hierarchy with *n* distinct directories is the similar, except for two differences. First, the entire hierarchy consists of directories rather than files. Secondly, rather than determining the depth of the hierarchy based on the number of files, the width of each level of the hierarchy is based on the desired depth of the hierarchy and the total number of directories. The same rule of allowing a maximum of 100 files or directories in a parent directory still applies.

Appendix D

fsbench: Flushing Caches

Caches are sized at the beginning of fsbench so that they can be flushed reliably during phases III and IV.

To flush the buffer cache, a file *n* times the estimated size of the buffer cache is written, and then read; *n* is the number of repetitions specified by the user, and defaults to five. Both the writes and reads are sequential. To flush the attribute cache, a hierarchy with *n* times the estimated size of the attribute cache distinct files is `stat`'d. And to flush the name translation cache, a hierarchy with *n* times the estimated size of the name cache distinct directories is `stat`'d. Note that the underlying assumption in all three cases is that whatever is read is put into the cache.

Appendix E

Aging Filesystems

To run the benchmark and the trace (Chapter 5) on a filesystem that might represent a real filesystem in use, the following two steps are taken. First, a logical snapshot of a filesystem (either from `ls -ilR` or constructed using the algorithms in Appendix B) is put on the disk by iterating through the tree: each directory is made and each file is created and initially written, according to the size parsed from the snapshot. Next, the filesystem is aged in the following manner. First, half the files are deleted in random order and then the same files are re-created in random order. Then, a quarter of the files, then an eighth of the files, down to one-sixty-fourth of the files. This process is repeated ten times.

To determine how aged the filesystem is, a utility written by Keith Smith calculates a fragmentation score based on the percentage of blocks optimally (*i.e.*, sequentially) allocated, and the length of the free extents left on disk. This utility only works under FFS. The statistics gathered are:

- Maximum number of contiguous blocks
- Filesystem block size
- Filesystem fragment size
- Rotational delay
- Number of free blocks
- Number of free fragments
- Total number of blocks allocated to data files and the total number of extents those blocks are in
- Average extent length of used blocks

- Percentage of used blocks optimally allocated
- Number of free blocks and the total number of extents those blocks are in
- Average size of a free extent
- Total number of blocks allocated and the total number of extents those blocks are in
- Average size of an allocated block extent

To determine how well this algorithm worked, the snapshot of an existing filesystem is put on a clean filesystem and aged. The resulting fragmentation score is calculated and compared to the score from the original filesystem. The results are presented in Table E.1.

Parameter	Original	Artificial
Blocks used (data only)	44674	44820
Number extents	20216	21905
Average extent length	2.21	2.05
Percentage of blocks optimally allocated	76.10%	75.62%
Free blocks	23492	87493
Number extents	1873	2533
Average extent length	12.54	35.54
Allocated blocks (data and meta-data)	46600	45332
Number extents	1879	2529
Average extent length	24.80	17.92

Table E.1 **Filesystem Aging Statistics**

Even though the disk used is approximately twice the size of the disk used for the original filesystem, the actual fragmentation of the artificial filesystem is comparable, if not worse. The only indication that the disk is larger is that the number of free blocks is much higher, and correspondingly, the average free extent length is larger too. This result is due to the fact that FFS tries to place files in the same cylinder group as their parent directory, thereby minimizing the number of extraneous cylinder groups that are touched. Note that this algorithm will both help and harm LFS performance. It will help LFS because all of the meta-data will be placed together and because all of the blocks in a file will be close together since files are created and deleted in their entirety. On the other hand, the meta-data may be nowhere near the corresponding data.