# Fstress: A Flexible Network File Service Benchmark

Darrell Anderson*
*Department of Computer Science*
*Duke University*
anderson@cs.duke.edu

## Abstract

Benchmarks provide repeatable workloads for testing and comparison. An ideal file service benchmark would be easy to use, be highly configurable to emulate arbitrary workloads, and scale to extreme load levels and data set sizes.

This paper introduces Fstress, a synthetic, flexible, self-scaling file service benchmark. We identify and support twelve workload-description parameters, enabling a wide range of workloads to evaluate file service scalability, sizing, configuration, and other factors. We demonstrate the generality of our approach by using these parameters to express seven popular existing workloads that exercise different aspects of a file service. As workload patterns continue to evolve, it is straightforward to capture and emulate their essential characteristics.

## 1 Introduction

David Patterson said, "For better or for worse, benchmarks shape a field." Then for better, benchmarks should represent the wide range of important applications, and must be able to adapt to new ones.

Fstress is a synthetic, flexible, self-scaling network file service benchmark. Our primary goal is flexibility, exporting control over both data set and workload, such as workload operation mix, file popularities, directory tree size and shape, etc. These "knobs" allow systematic testing and sensitivity analysis, or simple scalability evaluation.

Moreover, this flexibility enables comprehensive evaluation of file service implementations that was not previously possible. Fstress could help study file system layout, decentralized file service schemes, hierarchical storage systems, storage provisioning, and so on.

Fstress is a robust, general tool that measures the file service as a component of a complete system. It evaluates the file service under heavy load representing an aggregate of clients, independent of the behavior of any specific client configuration. Fstress has been available for more than a year, and is in active use by several storage companies.

This paper is organized as follows. Section 2 discusses file service characterization. Section 3 provides an overview of the Fstress "knobs" with their file service implications. Section 4 discusses the Fstress benchmark structure. Section 5.2 outlines included workloads, and we present experimental results in Section 6. Section 7 concludes.

## 2 Motivation and Related Work

File system benchmarks divide into four categories: real traces, macro-, micro-, and synthetic benchmarks. Traces have been used for workload characterization in the research community. Berkeley studies date back to 1985 with Ousterhout's analysis of the UFS file system [25], then Baker's 1991 Sprite measurements [3], Dahlin's 1994 study of an Auspex file server [11], and Roselli's 2000 analysis of three academic groups' file accesses [27].

Most of these traces were taken from computer science departments, featuring low I/O demands and workload patterns derived from document preparation and software development. The observed low load is insufficient for large file service analysis

*This work is supported by the U.S. National Science Foundation (CCR-00-82912, EIA-9972879, and EIA-9870728), Cisco Systems, and equipment donations from IBM.

(e.g., the Roselli Web trace records 2,400 accesses per day). Traces may be augmented, e.g. by applying them at an artificially high rate, but the disparity between authentic load and large-scale file service saturation may be several orders of magnitude.

Macrobenchmarks run real applications, such as building software packages from source, using total time as the figure of merit. These benchmarks are difficult to vary for sensitivity analysis, and are highly sensitive to client configuration.

Microbenchmarks test one aspect of a file service. For example, Bonnie [5] and Postmark [19] exercise bulk and small-file I/O respectively. Measuring one dimension is valuable for testing, but by definition microbenchmarks have a narrow focus [24]. For example, a single sequential I/O stream may perform well in isolation, but drastically worse when combined with a competing stream.

Synthetic workloads emulate characteristics observed in real environments. They are often self-scaling [7], augmenting their capacity requirements with increasing load levels. The synthetic nature of these workloads enables them to preserve workload features as the file set size grows. In particular, the SPECsfs97 benchmark [8] (and its predecessor LADDIS [21]) creates a set of files and applies a pre-defined mix of NFS operations.

SPECsfs97 tests one particular workload type. It offers limited flexibility, but changing parameters is cumbersome, complicating systematic study along a particular dimension (e.g., favoring namespace operations). It also has high overhead, requiring many clients to generate a significant load level.

The SynRGen synthetic benchmark [14] addresses rigid workload concerns by stochastically combining simple models of user tasks. For example, it represents a compile task by reading a "source" file and several "header" files, then writing an "object" file. This technique requires a rich set of task models and proper mixes to define a specific workload.

All of these characterization and benchmarking efforts emulate workload patterns from specific environments that may not match those for large servers today [24]. A file service benchmark should be flexible to model a wide range of workloads [31], as well as to adapt to new and changing demands.

Fstress is a synthetic, flexible, self-scaling file service benchmark. Like SPECsfs97, Fstress uses probabilistic distributions to govern workload mix and access characteristics. Beyond SPECsfs97, Fstress adds file popularities, directory tree size and shape, and other controls. Fstress includes several important workload configurations, such as Web server file accesses, to allow apples–to–apples comparisons over different systems, and to provide familiar workloads for testing a specific system.

## 3 Parameters and Workload Properties

This section describes key properties parameterizable in Fstress and discusses relevant file service design issues. Some properties interact to affect multiple design issues, such as file sizes and popularities: a small number of popular small files may fit in a file buffer cache, favoring large-memory servers, while many uniformly popular moderately sized files may overflow even a large cache, favoring I/O capacity.

We separate parameters into three classes. The first class affects the working set, created *a priori* to the testing phase. The second class controls run-time decisions during measurement. The last class, non-parameters, are outside the scope of the benchmark.

### 3.1 Creation Phase Parameters

The first class of parameters influences the set of files and directories created before and between measurements. We identify eight such parameters:

**File, directory, and symlink counts**: When creating or augmenting a directory tree, these distributions control the number of files, directories, and symlinks created within each new directory. The shape and the distribution of directory sizes determines the effectiveness of name space management, and its importance. For example, hash-based lookup schemes [18] would likely outperform linear search for large directories.

**Maximum directory tree depth**: We create and augment the directory tree recursively. This parameter bounds the depth and implicitly limits the maximum increase in file set.

**File, directory, and symlink popularity**: During the measurement phase we choose operation targets according to popularity. These access probability distributions control the popularity skew for newly created objects. Popularity and temporal locality for files and directories determine the importance of file and name caching. If accesses skew toward a small number of popular objects, then file servers may benefit from a higher ratio of memory and processing power to disk storage; unpopular objects "pollute" the file service's store.

**File sizes**: The ratio of small to large files influences whether a workload is data- or metadata-intensive. Metadata-intensive workloads skew toward name space operations. These are more likely in Internet service environments involving mail, messaging boards, or Web caching. If large files are common, such as in database environments, it may be beneficial (or even necessary) to partition individual files across multiple subsystems.

### 3.2 Measurement Phase Parameters

Benchmarking proceeds after creating/augmenting the data set. The second class of parameters controls run-time behavior during this measurement phases. We support four run-time parameters:

**Operation mix**: Our benchmark chooses request types from this distribution. The operation mix affects file service performance in many related ways. Operations such as directory creation and removal, *readdir* directory scans, *rename*, and *link* operations place pressure on the name space. Also, the ratio of asynchronous *writes* to *commit* (fsync, or on close) operations determines the cost of coordinating *commits* if a volume is distributed across multiple subsystems. If *commits* are frequent, structures that place each file on a single subsystem may offer better write performance by reducing the number of devices to stabilize.

**I/O read and write sizes**: Once we decide to perform an I/O operation, we must choose the amount of data to read or write. I/O patterns motivate different data placement and prefetching schemes. If large, sequential I/O is common the file service might benefit from deep prefetching for reads, and ample buffering to gather writes. Conversely, sequential prefetching wastes bandwidth for random or sequential-but-short I/O patterns.

**Load level**: Fstress generates requests at a specified fixed rate. We combine results from a series of experiments to evaluate response to increasing load, augmenting the request generation rate until file service saturation.

### 3.3 Non-Parameters

The last class comprises features we chose not to support. We discuss why we excluded these aspects, and how their omission affects the value of this benchmark. In some cases, we discuss how missing features could be added.

**Protocol**: Fstress currently supports only NFS version 3. NFS is deployed widely and implementations exist for most file servers. Fstress encapsulates protocol handling to simplify adding others; a DAFS [20] Fstress module is under development.

**Inter-arrival time**: Fstress issues requests at a uniform rate. It could use other policies, however at high load levels at the generator queuing in the client socket buffers and other system artifacts may disrupt or introduce short bursts into the timing scheme (including our uniform rate).

**Connection set size**: To enable various high-intensity load generation optimizations (see Section 4), each Fstress load generating client opens a single connection to the file server. Large installations that handle high load levels are likely to use hundreds or perhaps even tens of thousands of connections, stressing connection management. While connection management is an important issue, we believe its performance influence is minor in comparison to handling the rest of the file service.

**Request dependencies**: Every Fstress request is independent of every other request except for sequential I/O series. Other dependencies, such as file *lookup–truncate–write* sequences or associating groups of files to be accessed together, require deeper control over operation types and targets. We chose one request/response as our unit of work. By contrast, SynRGen [14] "operations" are sophisticated tasks, defining a series of related actions.

These aspects describe our evaluation domain of

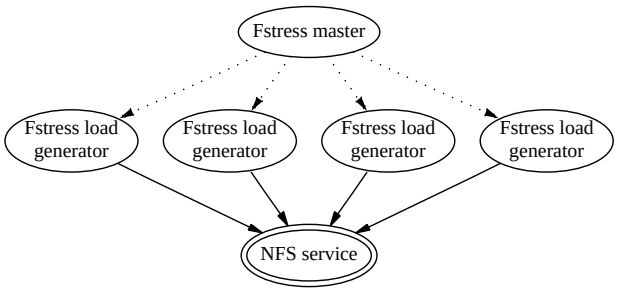


Figure 1: Fstress control structure.

aggregate workload characteristics. They enable a simple and efficient benchmark structure.

## 4 Benchmark Structure

The Fstress structure enables high-intensity request generation with accurate timing properties. It generates load at a fixed request rate against an NFS server, measuring operation latencies. Maximizing per-client load generation capacity is important to reduce both the number of clients as well as switch ports necessary to saturate high-end file servers.

Fstress reports per-operation latencies, success, error, and timeout counts. It also generates a SPECsfs97-like "figure of merit" [8], the average overall request latency (with standard deviation). Typically, it combines many runs to draw a latency versus load graph, to and past saturation.

Fstress uses a simple master/slave control structure, depicted in Figure 1. The master process invokes and coordinates remote slave processes, with one process per load generating client.

Fstress consists of a set of components, layered in two halves. Generic benchmark, measurement, object and distribution sets, and event framework modules comprise the "top half." The "bottom half" ties to the system, in this case raw socket ONC-RPC, NFS encoding and decoding (XDR), and precision timing.

The bottom half uses a single-process, open-loop event model centered on a socket poll designed to spread load generation smoothly across time, maintaining constant load even when the measured system gets slower, e.g. at saturation. Requests are asynchronous, with continuation functions called

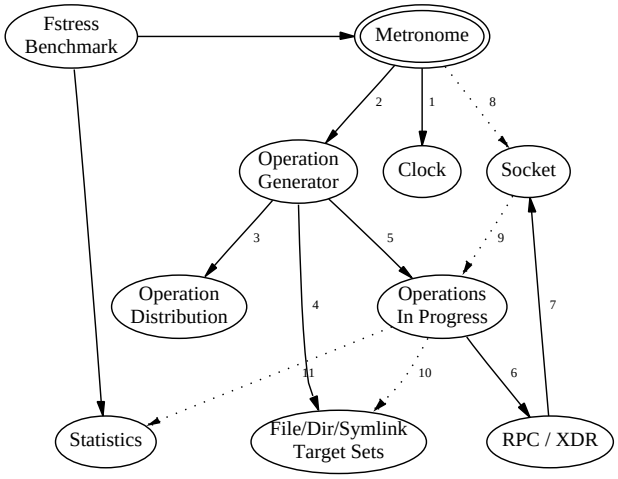


Figure 2: Fstress load generator structure.

when responses arrive or when requests time out.

This model avoids scaling problems associated with large numbers of outstanding requests. First, it multiplexes all operations on a single socket to avoid issues associated with *select* over large sets [4]. Second, by using a single process, it escapes scheduling delays, timing errors, and bursty behavior due to legions of runnable client processes [16].

Fstress relies on clock precision to measure operation latencies and to meter request generation. Our clock module provides low overhead, microsecond time granularity by supplementing a cached *gettimeofday* result with the processor cycle counter.

Figure 2 illustrates key components and control flow within the Fstress load generator. Arrows denote connections between modules, ordered to follow a typical request/response sequence. Solid arrows follow a request generation, and dotted arrows trace the response. Numbers mark specific actions described below.

After initializing or augmenting the file set, Fstress drops into its "metronome" event loop. The metronome generates requests at a regular rate, services responses, and retransmits aging requests if using an unreliable transport layer. The metronome checks the clock (1) to see if it needs to generate new requests (2) to meet the desired request rate. If so, it selects a request type from the operation mix (3), then a file, directory, or symlink target from

the object sets (4) described in detail in Section 4.1. After entering the request into the operations-in-progress set (5), it generates the wire request (6) and sends it through a raw socket (7).

In addition to generating requests, the metronome loop polls the socket for incoming replies (8). It matches replies against the outstanding operations set (9), then applies side effects to the target object (10) such as increasing file size after an extending write. Finally, Fstress records the operation latency and success value into a statistics module (11).

### 4.1 Object Sets

Fstress maintains file, directory, and symlink information in compact, weighted (by popularity) object sets. We want fast access and selection, with low space overhead memory-resident structures and an efficient weighted-selection algorithm. We developed a simple weighted "sparse heap," data structure using heap-like child/parent association by index instead of pointers. Unlike ordered heaps, free entries intermingle with valid entries, hence the "sparse" nature. Objects remain at fixed locations, simplifying inter-object and outside references.

Each entry in the object set contains its weight and the aggregate weight of the subtree rooted at its location. Free entries have zero weight, but still maintain the weight of their subtrees. For $n$ objects, weighted selection runs in $O(\log n)$ time by choosing a uniform random number between 0 and the total tree weight, and traversing down to find the object whose weight (when combined with its left subtree) contributes the random value. This structure allows flexible weight distributions in the face of object insertion and deletion.

## 5 Workload Variants

Fstress provides a single, consistent tool to test and evaluate file services under a variety of workloads. Fstress defines workloads by a series of parameter distributions as outlined in Section 3. Distributions are histograms contained in text files following a simple *value:weight* syntax. The user may modify files by hand, or (re)generate them using other Fstress tools that help to adjust along one or more dimensions (e.g., changing a popularity skew, read/write ratio, etc.).

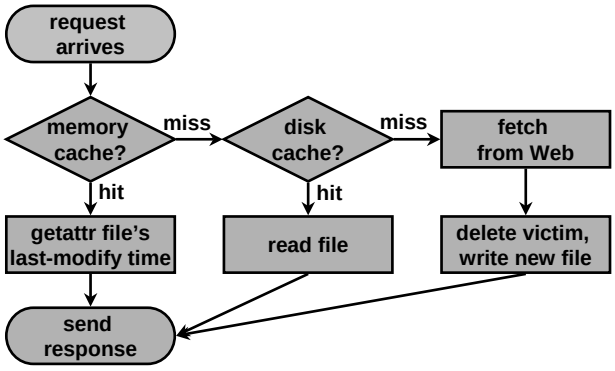


Figure 3: Simple Web proxy model.

### 5.1 Workload Estimation

This section outlines a sample workload estimation for a Web proxy based on the simplified model of proxy actions illustrated in Figure 3. When this proxy receives a request, it first queries its memory cache, sending the response if present. If absent from the memory cache, it probes its disk cache. If present, it reads it into its memory cache then sends the response. Finally, if absent from both the memory and disk caches, it retrieves the object from the Web, writes it to its disk cache (deleting a replacement candidate if full), then sends the response.

The top row in Figure 3 reflects control flow, the second lists file system effects. If a request hits in the memory cache, our model checks the file's last-modify time with a *getattr* request before sending the response. Likewise there are similar consequences for other actions. Given this model and estimates for memory and disk cache hit rates (e.g., from [22]), we can compute an overall operation distribution. File size and popularity distributions are also given in the literature ([2, 26, 10]).

Workload descriptions can be extracted from traces, but model-based derivation provides a useful high-level abstraction of what an application is really doing. By exposing controls at this level, one may perform high-level sensitivity studies such as varying proxy memory or disk cache hit rates. Trace results could be used to validate a model-based workload.

| workload | file popularities | file sizes | dir sizes | I/O accesses |
|---|---|---|---|---|
| SPECsfs97 | random 10% | 1 KB – 1 MB | large (thousands) | random r/w |
| Web server | Zipf ($0.6 < \alpha < 0.9$) | long-tail (avg 10.5 KB) | small (dozens) | sequential reads |
| Web proxy | Zipf ($0.6 < \alpha < 0.9$) | long-tail (avg 10.5 KB) | small (dozens) | sequential r/w |
| transaction processing | few files | large (GB - TB) | small | random r/w |
| peer–peer media server | Zipf (large $\alpha$) | large (avg 3.7 MB) | large (100 – 1000) | sequential r/w |
| mail server | Zipf ($\alpha = 1.3$) | long-tail (avg 4.7 KB) | large (500+) | seq r, append w |
| news server | Zipf (small $\alpha$) | bimodal (5 or 45 KB) | large (100+) | random r, append w |

Table 1: Some workload types.

| | SPEC 1997 | Web server | Web proxy | data-base | peer-peer | mail server | news server |
|---|---|---|---|---|---|---|---|
| lookup | 27% | 14% | 14% | 0% | 1% | 27% | 1% |
| read | 18% | 28% | 6% | 61% | 54% | 14% | 22% |
| write | 9% | 0% | 23% | 31% | 35% | 24% | 64% |
| getattr | 11% | 55% | 18% | 3% | 1% | 3% | 0% |
| readlink | 7% | 0% | 0% | 0% | 0% | 0% | 0% |
| readdir | 2% | 1% | 1% | 0% | 0% | 0% | 1% |
| create | 1% | 0% | 11% | 0% | 1% | 0% | 1% |
| remove | 1% | 0% | 11% | 0% | 1% | 0% | 1% |
| mkdir | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| rmdir | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| fsstat | 1% | 1% | 1% | 0% | 1% | 1% | 1% |
| setattr | 1% | 0% | 0% | 0% | 0% | 4% | 0% |
| readdirplus | 9% | 0% | 0% | 0% | 0% | 0% | 0% |
| access | 7% | 1% | 4% | 1% | 1% | 3% | 1% |
| commit | 5% | 0% | 12% | 4% | 5% | 24% | 8% |

Table 2: Sample workload operation distributions.

### 5.2 Fstress "Canned" Workloads

This section describes the "canned" workloads included with Fstress, and discusses their corresponding file service interactions. Tables 1 and 2 summarize the workloads with their important sizing parameters and operation mixes, respectively. We derive values from studies in the literature (as cited in each section below).

Our descriptions define an instance of a particular workload. In practice workloads may vary with internal factors such as client file system cache sizes, or external ones like file popularity. We outline prominent factors and effects for each workload.

These workload descriptions are based on published studies of real workloads in a diversity of settings. Validating each workload is beyond the scope of this paper; we do not have access to all systems modeled, nor traces to strengthen our estimations. We expect some inaccuracy in our models, and provide them as plausible configurations within the wide range of possible workloads. Upon deriving a better workload estimation, one need only set the Fstress knobs appropriately to emulate it.

**SPECsfs97**: The Standard Performance Evaluation Corporation introduced their System File Server benchmark (SPECsfs) [8] in 1992, derived from the earlier self-scaling LADDIS benchmark [21]. SPECsfs used 1987 studies at Sun Microsystems to determine the operation mix and file size distributions. The second release (1997) added support for NFS version 3 and updated the workload mix to reflect surveys of over one thousand NFS version 2 servers (NFS version 3 had not yet been widely deployed). A recent (2001) revision corrected several defects identified in the earlier version [16].

**Web server**: Several efforts (e.g., [2]) attempt to

identify durable characterizations of the Web. Web objects follow a Zipf popularity distribution [26], where object $i$ has a probability of access proportional to $1/i^{\alpha}$, with $\alpha$ typically between 0.6 and 0.9. File sizes appear long-tailed [10]. In the March 5, 2001 NLANR trace [1], the average file size is 10.5 KB, median 1.3 KB, and maximum 138 MB.

Requests coming into a Web server manifest themselves differently to the underlying file system due to a caching "trickle effect" [12]. Even a small in-memory cache (either at application level, or as a file system buffer cache in front of a network file service) may satisfy a large portion of requests to popular objects. The dominant file system operation may be *getattr*, verifying that a cached file has not changed before sending it.

The operation mix reflects this phenomenon. A hit in the memory file cache will likely also hit in the system name cache, generating only a single *getattr* file system request. A miss, on the other hand, must first *lookup* the file, then issue *read* calls until exhausting the file. The model uses an 80% hit rate [2] and an average of two 8 KB *read* requests to read a file. It reads files sequentially and in their entirety. The Web workload is read-only.

The ratio of I/O to metadata requests varies with cache and object sizes, and popularities [12]. Bigger caches or smaller object sizes produce more cache hits. Also, hit rate may increase if the popularity skew reduces the working set size (larger $\alpha$).

**Web proxy**: From a file system perspective, Web proxies are essentially Web servers that occasionally (or frequently, depending on the miss rate) write new files and delete old ones [22]. Just as with the Web server case above, a Web proxy hit manifests itself as either an attribute check (*getattr*) if the file is already in a memory cache, or a *lookup* and sequential read if it is not. A Web proxy miss generates a *create* and sequential file write, with occasional *remove* to enforce capacity restrictions.

Like the Web server, the reads/attribute check ratio follows the server memory miss ratio dependent on memory size and object popularities. Unlike the Web server, the Web proxy creates and deletes files. The ratio of writes to the combination of reads and attribute checks follows the proxy cache miss ratio, typically between 19% and 55% [13].

**Transaction processing**: For performance reasons, database designers often bypass file systems, instead accessing data through raw devices. With faster systems, more database users are leveraging file servers to ease administrative costs [17].

We model our transaction processing workload after TPCC [9], reading and writing within a handful of large files in a 2:1 ratio. I/O access patterns are random, with some short (256 KB) sequential asynchronous writes with *commit* (fsync) to mimick batch log writes.

**Peer-to-peer server**: In peer-to-peer file sharing, users cooperate sharing storage and network resources in order to exchange files. A study of Napster and Gnutella [30] shows two user classes. The majority of users act like clients or "free riders," accessing the service but exporting few files themselves. The second, smaller (7% of hosts) class acts like servers, each hosting thousands of files.

We model the server class due to its higher load and storage capacity demands. Downloads, retrieving a file from elsewhere in the service, follow a *create* and sequential asynchrous *writes* with *commit* (fsync). Uploads, sending a file to another client, act as a *lookup* with sequential *reads*. Other actions, *readdir* and *remove*, model scans and user-driven file deletion. For this class of server, uploads are 50% more frequent than downloads.

For this study, multimedia files dominate with an average file size of 3.7 MB. Due to this large file size, I/O requests are two orders of magnitude more frequent than namespace operations.

**Mail server**: Electronic mail servers frequently handle many small files, one file per users' mailbox. Servers append incoming messages, and sequentially read the mailbox file for retrieval. Some users or servers truncate mailboxes after reading.

The workload model follows that proposed by Saito et. al. [28], with mail delivery accounting for 90% of mail server transactions, and mailbox retrieval for the remaining 10%. In both cases, it uses a Zipf-like popularity distribution with $\alpha$ set to 1.3. The average message size is 4.7 KB, with a long-tail (Pareto distribution) out to 1 MB.

| Server | Memory | Disks |
|---|---|---|
| BASELINE | 256 MB | 1 disk |
| DISK+ | 256 MB | 4 disks |
| DISK++ | 256 MB | 7 disks |
| MEM+ | 1 GB | 1 disk |
| MEM++ | 1.75 GB | 1 disk |

Table 3: File server configurations.

**News server**: Usenet provides public discussion forums (newsgroups) on a wide and varied range of topics organized into shallow hierarchies. An analysis of a large news server [29] shows a near-flat popularity distribution, with no group comprising more than 2.5% of accesses. Despite the large user population studied, many messages are never read and less than 5% of messages are read more than three times (from one server). File sizes follow an interesting bimodal distribution, showing two common file sizes near 5 KB and 45 KB for text and binary image oriented newsgroups, respectively.

Newsgroups map into a series of files, one per newsgroup. New messages append to the end of files, and users read random sections therein.

## 6 Experimental Results

In this section we present results from several Fstress experiments. We demonstrate Fstress use, measuring file server scaling and quantifying when and how much additional resources help. We test the scalability of the benchmark itself, and also show how turning workload parameter "knobs" can be used to study a file service. The intent is to show the utility of the Fstress workload models and flexibility, as well as the overall value of the benchmark to evaluate file services.

### 6.1 Experimental Setup

We use five file servers with different memory and I/O capacities, summarized in Table 3. Starting with a BASELINE server with small file cache and one disk, we add resources along two different dimensions for the remaining servers. We augment the file cache sizes (but keep a single disk) for the MEMORY servers, and likewise fix the file cache size but add more disks to the DISK servers.

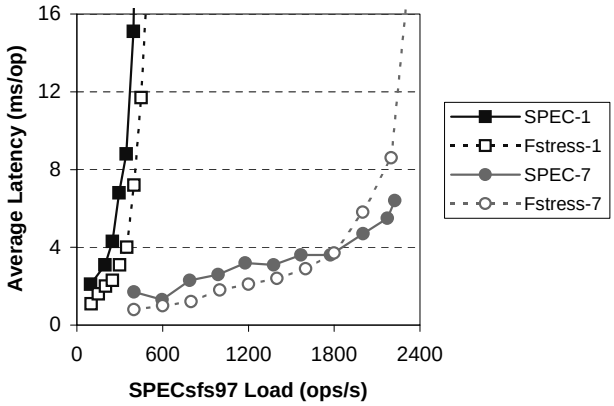


Figure 4: Fstress mimicking SPECsfs97.

The BASELINE and DISK servers are Dell PowerEdge 4400s with 733 MHz Pentium-III Xeon CPUs, 256 MB RAM, and eight 18 GB 10,000 RPM Seagate Cheetah drives connected to two dual-channel Ultra-160 SCSI controllers. The BASELINE server exports a single disk, and the DISK servers combine and export four and seven disks, respectively, as single volumes with the *ccd* striping concatenated disk driver.

The MEMORY servers are Dell PowerEdge 1550s with a 1 GHz Pentium-III and an 8 GB Seagate Cheetah drive on an Ultra-160 SCSI controller. They have 1 GB and 1.75 GB RAM respectively.

Servers run FreeBSD 4.3 exporting NFS over UDP with 20 *nfsd* processes. Each exports a single UFS/FFS volume configured with soft updates [15].

Load generator clients are Dell PowerEdge 1550 rackmounts with 1 GHz Pentium-IIIs. All systems have Gigabit Ethernet NICs connected by an Extreme Summit 7i switch. Except where otherwise noted, we use four load generators. For each data point Fstress warms the servers for two minutes at the desired load level, then measures for four minutes. It augments the data set between tests as appropriate to the workload.

### 6.2 Validation with SPECsfs97

Flexibility is a primary Fstress design goal. This flexibility lets us configure Fstress to behave like SPECsfs97, an industry-standard self-scaling benchmark [8] described in detail in Section 5.2. Once configured, we compare it with

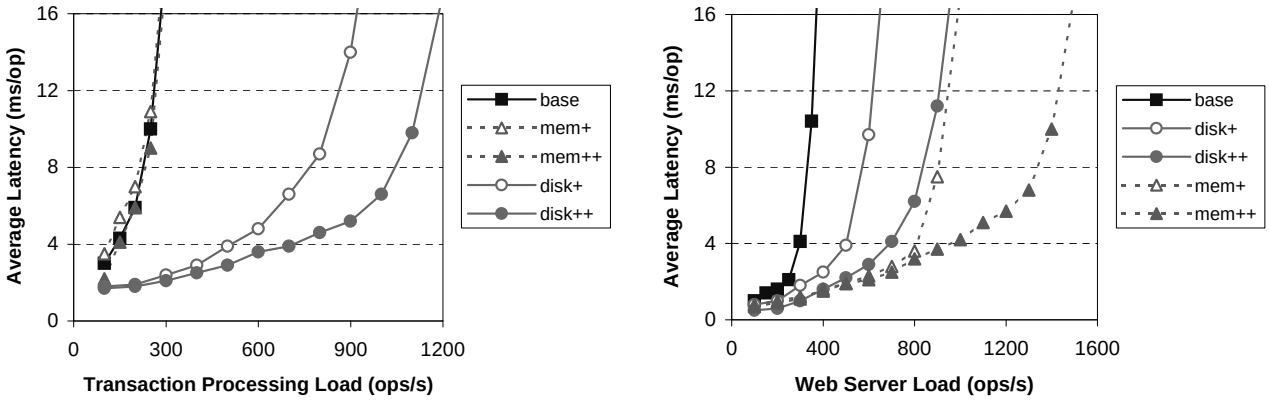


Figure 5: Evaluating server resource utility for different workloads.

real SPECsfs97 results from the same experimental setup. We plot these results in Figure 4, measuring operation latency with respect to a SPEC-like offered load. The two server configurations here reflect exporting one ("1") or seven ("7") disks (unified into a single striped volume), the BASE and DISK++ servers.

We plot two lines for each configuration: a solid line for SPECsfs97, and a dotted line for Fstress SPEC-like mode. For the 1 disk configuration, latency grows with offered load until the server saturates near 400 NFS ops/s. The 7-disk test behaves similarly, saturating at 2,200 NFS ops/s. In each case, Fstress generates operations within 0.1% of the desired operation mix.

Both benchmarks yield comparable results and saturation points. Both exhibit sharp latency spikes at saturation for the 1 disk configuration, however only Fstress shows this for the 7 disk setup. SPECsfs97 limits its maximum number of outstanding I/Os, throttling back request generation instead of exceeding this limit. As a result, it mitigates request queuing at the server and corresponding higher latencies.

### 6.3 Evaluating Disparate File Servers

In this section, we describe Fstress results measuring the five different servers. Intuitively, the MEMORY servers perform well for working sets that fit in their file cache, and poorly for overflow due to their single disk arm. Conversely, the DISK servers' small file cache hinders their ability to handle large working sets from memory, but their multi-armed I/O subsystem copes well, especially for very large cache-defeating working sets. We choose two workloads appropriate to this disparity, and compare the results.

For the first experiment, we use Fstress to apply the transaction processing workload described in Section 5.2. The left graph in Figure 5 plots observed latency as a function of offered load against each of the five servers. There is a clear disparity between single and multi-disk servers; the database's multi-gigabyte files and random I/O patterns defeat even large file caches. Increasing the file cache size has negligible effect on performance, while the extra disk arms of DISK+ and DISK++ scale to much higher load levels.

The second experiment follows the same methodology as the first, except substituting the Web server instead of the transaction processing workload. As described in Section 5.2, this workload is read-only and its file accesses follow a Zipf distribution. As a result, the working set is amenable to caching.

The right graph of Figure 5, again plots observed latency as a function of offered load, this time using the Web server workload. Unlike the previous experiment, each enhanced server delivers some performance benefit. Fixing the file cache size and increasing the number of disks yields a modest benefit. The DISK+ server handles nearly double the load of BASELINE, and DISK++ handles twice again as much.

In this experiment, file cache size has a more significant effect than I/O capability. The MEMORY+

server with its single disk handles slightly more load than DISK++, and MEMORY++ again almost doubles the saturating load level. This is due in part to requests queuing in the servers, as well as increase in file cache miss rates as the (self-scaling) working set grows.

From a storage system builder's perspective, the transaction processing workload does not benefit from file cache, but scales to higher load levels with more disk arms. Conversely, the Web server workload sees only a modest impact with more disk arms, but shows a significant improvement from a larger file cache. A single-workload benchmark generates only one result. By providing both results (and the ability to test others), Fstress helps identify and evaluate cost/performance tradeoffs along multiple dimensions.

### 6.4 Web Scrutiny

In addition to the high-level latency-versus-throughput results graphed above, Fstress reports detailed per-operation breakdowns and statistics. These results come from Fstress load generator client logs, with similar results observed between peer clients for each experiment. Table 4 shows such a subset of the breakdown for one data point in the Web server graphs. We choose 600 ops/s and list results for two servers with comparable latencies at that load: MEM+ and DISK++. The average latency is 2.22 ms/op for the MEM+ server, and 2.18 ms/op for DISK++.

The per-operation response times are similar for both servers, with slightly faster *lookup* and *read* for the MEM+ server probably due to more requests served from the file cache. The latency standard deviations are much higher for MEM+ than DISK++. The MEM+ server has a low average latency, but with high variance because requests that require disk I/O queue for that constrained resource. The DISK++ server delivers more consistent latencies, with lower standard deviations.

For clarity, Table 4 omits additional per-operation results recorded in the Fstress logs. For each operation type, Fstress also counts the total number of requests attempted, retransmissions, error results, and cancellations (operations are canceled after exceeding a configurable number of retransmissions). Between both experiments, there were 5 retransmissions and no errors. These counts increase as servers saturate.

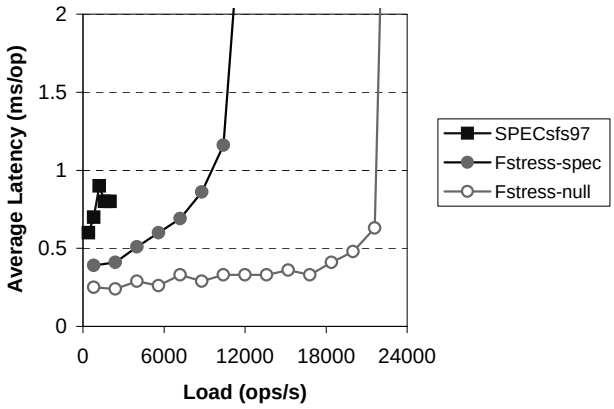


Figure 6: Single Fstress load generator capacity.

### 6.5 Benchmark Scalability

Load generators are user-level programs, relying on the underlying operating system and network interface to carry requests. Maximizing per-client load generation capacity is important to reduce both the number of clients as well as switch ports necessary to saturate high-end file servers.

In Figure 6 we measure the capacity of one load generator. We test against a modified kernel-based NFS server that responds success to every request without any real file service behind it (zero-filling reads). This server handles extreme request rates with low latency, enabling client saturation tests.

We plot two results from Fstress, and for comparison, we also measure the SPECsfs97 benchmark, showing high latencies even at low load levels. A single SPECsfs97 client is unable to generate load beyond 2,000 requests per second. At this rate, SPECsfs97 observes roughly double Fstress latency, polluting its results.

The first Fstress result uses Fstress configured to generate a SPEC-like load. One Fstress client saturates at about 12,000 requests per second, and can generate up to 10,000 requests per second with the same latency as SPECsfs97 at one-fifth the load. The final line measures a "null" workload comprised entirely of NFSPROC_NULL requests, an

| NFS op Type | Mix % | MEM+ Response Time (ms) | MEM+ Std Dev (ms/op) | MEM+ % Total Time | DISK++ Response Time (ms) | DISK++ Std Dev (ms/op) | DISK++ % Total Time |
|---|---|---|---|---|---|---|---|
| lookup | 14% | 1.8 | 29.6 | 11.0% | 2.9 | 8.6 | 18.6% |
| read | 28% | 3.9 | 45.5 | 49.4% | 4.9 | 10.9 | 62.9% |
| getattr | 55% | 1.5 | 28.3 | 37.6% | 0.7 | 2.6 | 16.8% |
| fsstat | 1% | 0.7 | 8.4 | 0.4% | 0.7 | 3.4 | 0.3% |
| readdir | 1% | 2.8 | 32.1 | 1.2% | 2.2 | 5.9 | 1.1% |
| access | 1% | 0.8 | 13.8 | 0.4% | 0.7 | 2.3 | 0.3% |

Table 4: Detailed performance breakdown for the Web workload at 600 ops/s over UDP.

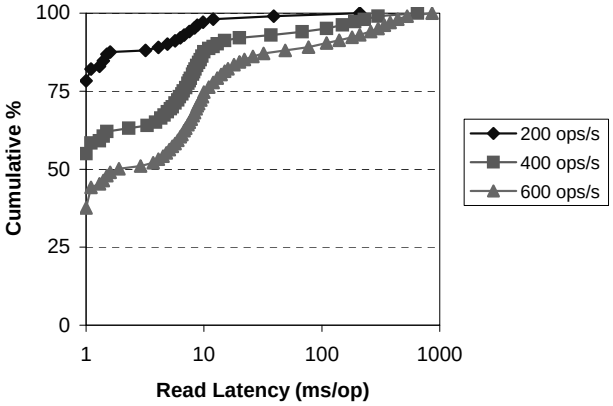


Figure 7: Self-scaling effects for read latencies.

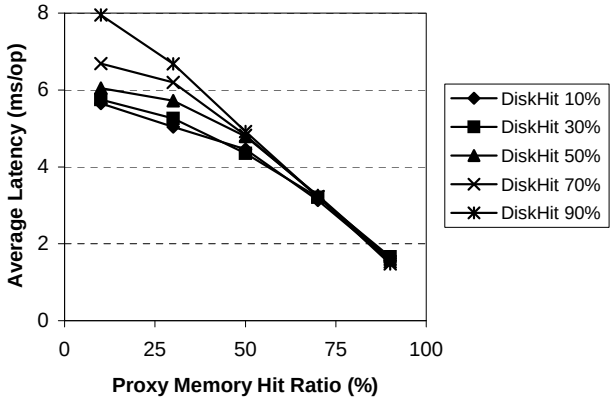


Figure 9: Varying Web proxy hit ratios.

RPC-level *ping*. This tests uses short requests and responses, minimizing bandwidth requirements and per-packet overheads. A single client can issue roughly 20,000 such requests per second.

### 6.6 Self-Scaling Consequences

Figure 7 demonstrates another Fstress analysis method, plotting Web server workload read latencies from the pant-2 file server described in Section 6.8. The three lines plot cumulative read latencies at three load levels from light to nearly saturated. Because Fstress is a self-scaling benchmark, as it increases the request rate, it also increases the file set size. The log scale emphasizes the consistent shape as well as behavior at low latencies. Each line exhibits different starting percentages, corresponding to requests satisfied directly from the server's file cache (response time less than 2 ms). At higher load levels, the larger working set size yields more misses in the NFS server file cache, decreasing the number of requests satisfied at these low latencies.

The next observation is a high density of requests satisfied near the 10 ms mark, corresponding to disk access time. After this cluster, we see long, steady growth until all requests are satisfied. Unlike average latencies, this result enables quality of service metrics such as the percentage of requests satisfied in less than 100 ms. For the low load, 99% of requests complete in under 100 ms. At the high load, only 89% do so as disk queues grow.

### 6.7 Varying Workload Parameters

An Fstress workload is a series of distribution files used to control the file set and runtime behavior of the benchmark. Workload descriptions follow various distributions that the user may vary to study file system behavior. For this experiment, we apply the Web proxy workload described in Section 5.2 against the DISK++ server at a fixed rate of 1000 requests per second. Figure 9 plots the results.

We vary two levels of hit ratios as illustrated in Figure 3. The first level, "Proxy Memory Hit Ratio" on the X axis, reflects the proportion of proxy cache

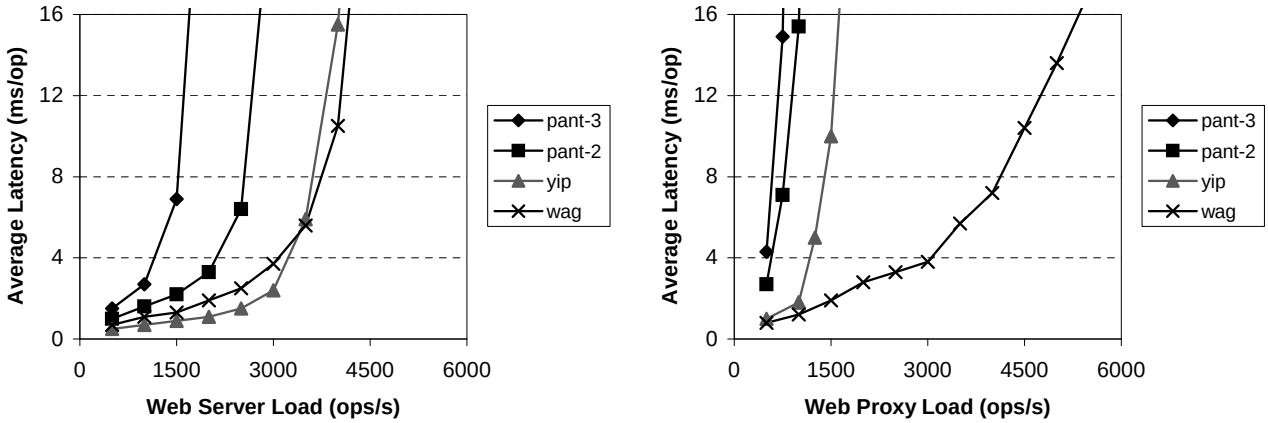


Figure 8: Large servers under read-only and read-write workloads.

in-memory hits. To serve a memory hit, the proxy workload issues only a *getattr* request to check the file timestamp. For a memory miss, it probes the file server with a *lookup* request, then either *reads* or *writes* to mimic file presence or absence, respectively. The lines plot different second-level hit rates (labeled "DiskHit-X%").

The workload shifts from data-intensive at the left side of the graph, reading and writing files, to metadata-intensive at the right, favoring timestamp checks. There are two trends visible in this graph. First, average latency decreases as memory hit rate goes up. This is due to the faster *getattr* operations, compared with *read* or *write*. While the FreeBSD name/attribute and file caches satisfy some of each type of request, *read* and *write* may require an additional disk operation per request compared to *getattr* due to reading or updating a file's UFS/FFS inode along with file data [23].

The second, less pronounced trend shows that average latency increases with the disk hit rate. The Web proxy workload uses asynchronous *writes* with later *commit* (fsync) to flush the data. As a result, *write* requests are faster than *reads*. Sequential prefetching on the server helps *read* latency somewhat, but with an average file size of 10.5 KB most files are too small to trigger it.

### 6.8 High-end Server Results

In this section we compare three higher-end servers. Because the servers differ in numerous ways, we limit evaluation to the Web server and Web proxy workloads. The read-only and read-write nature of these workloads let us judge how different servers react to an I/O type shift. The three servers are:

**Pant-2,3**: Dual 800 MHz processor IBM TotalStorage Network Attached Storage 200 server with 900 MB RAM running Linux v2.4.17-smp. This machine contains fourteen 36 GB IBM Ultra-160 10,000 RPM disks connected by an IBM ServeRAID-4h adapter configured to RAID-5. Pant exports a 434 GB test volume via Gigabit Ethernet, configured to EXT2 [6] and EXT3 [32] for pant-2 and pant-3, respectively.

**Wag**: Network Appliance F820 filer with fourteen 72 GB 10,000 RPM disks and 128 MB of non-volatile RAM. We test a 100 GB WAFL [18] volume over 7 disks, exported via Gigabit Ethernet.

**Yip**: This is a two-stage server. The back-end 105 GB test volume resides on a 3.2 terabyte IBM Shark connected to a front-end by a 100 MB/s FibreChannel link. The front-end is a 1 GHz Pentium III Linux v2.4.16 server with 256 MB RAM. It mounts the back-end as a local block device via a Qlogic ISP2200 host adapter, manages it as an EXT2 file system, and re-exports it over 100 Mb/s Ethernet.

Figure 8 plots results from this experiment, with Web server results on the left and Web proxy results on the right. For the Web server test, the three servers have similar shapes, each demonstrating gradual saturation. Each outperforms the best web server (MEM++) from Figure 5.

The read-write Web proxy load exposes some inter-

esting differences. First of all, the EXT2 volumes saturate at roughly half the request rate compared to the similar read-only Web server workload. The NetApp server exhibits a fairly gradual increase in latency as load increases (almost linear after 4,000 ops/s). The Network Appliance WAFL file system uses a flexible write placement policy, requiring fewer and shorter disk seeks than EXT2. EXT3 adds journaling support to EXT2, however it actually performs worse than EXT2 in its default configuration (the primary benefit from EXT3 is rapid recovery after failure). We also attempted this experiment with ReiserFS and JFS, but each locked up the server at high load levels.

## 7 Conclusions

This paper introduces Fstress, a flexible network file service benchmark. Fstress provides efficient, scalable, high-intensity load generation. In the spirit of Mogul's plea in [24], Fstress allows systematic exploration of a file service workload parameter space, as well as evaluation under several "real world" workloads. It presents high-level measurements useful for gross systems comparison and scalability tests, and detailed performance breakdowns to aid design and configuration.

We represent a series of real-world workloads, and apply them to demonstrate the importance of flexibility to evaluate disparate file servers. We show the scaling effects from changing server file cache size and I/O capacity, as well as from intra-workload shifts from data to metadata operations.

## 8 Availability

The Fstress synthetic benchmark with C source code (for FreeBSD, Linux, and Solaris) and additional documentation are freely available from the project web page: `http://www.cs.duke.edu/ari/fstress`.

## References


[1] National laboratory for applied network research (NLANR). `http://moat.nlanr.net`.

[2] Martin Arlitt and Carey Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 126–137, April 1996.

[3] Mary Baker, John Hartman, Michael Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, October 1991.

[4] Gaurav Banga, Jeffrey Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–265, June 1999.

[5] Tim Bray. Bonnie file system benchmark, 1996. `http://www.textuality.com/bonnie`.

[6] Rémey Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Dutch International Symposium on Linux*, 1986.

[7] Peter Chen and David Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performence. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, May 1993.

[8] Standard Performance Evaluation Corporation. SPEC SFS release 3.0 run and report rules, 2001.

[9] Transaction Processing Performance Council. TPC benchmark C standard specification, August 1992. Edited by François Raab.

[10] Mark Crovella, Murad Taqqu, and Azer Bestavros. *In A Practical Guide To Heavy Tails*, chapter 1 (Heavy-Tailed Probability Distributions in the World Wide Web). Chapman & Hall, 1998.

[11] Michael Dahlin, Clifford Mather, Randolph Wang, Thomas Anderson, and David Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[12] Ronald Doyle, Jeff Chase, Syam Gadde, and Amin Vahdat. The trickle-down effect: Web caching and server request distribution. In *Proceedings of the Sixth International Workshop on Web Caching and Content Delivery*, June 2001.

[13] Bradley Duska, David Marwood, and Michael Feeley. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[14] Maria Ebling and M. Satyanarayanan. SynRGen: An extensible file reference generator. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[15] Greg Ganger, M. Kirk McKusick, Craig Soules, and Yale Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[16] Stephen Gold. Defects in SFS 2.0 which affect the working-set, July 2001. `http://www.spec.org/osg/sfs97/sfs97_defects.html`.

[17] Network Appliance Enterprise Storage Group. NetApp filers feed the database engine, January 2001. Business brief, `http://www.netapp.com/tech_library/3101.html`.

[18] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Annual Technical Conference*, pages 235–246, January 1994.

[19] Jeffery Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, October 1997.

[20] Jeffrey Katcher and Steve Kleiman. An introduction to the Direct Access File System. Technical report, Network Appliance, June 2000.

[21] Bruce Keith and Mark Wittle. LADDIS: The next generation in NFS file server benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, pages 111–128, June 1993.

[22] Evangelos Markatos, Manolis Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary storage management for Web proxies. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999.

[23] M. Kirk McKusick, William Joy, Samuel Leffler, and Robert Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[24] Jeff Mogul. Brittle metrics in operating systems research. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems*, March 1999.

[25] John Ousterhout, Hervé Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace-driven analysis of the UNIX file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.

[26] Chris Roadknight, Ian Marshall, and Debbie Vearer. File popularity characterisation. In *Proceedings of the 2nd Workshop on Internet Server Performance*, May 1999.

[27] Drew Roselli, Jacob Lorch, and Thomas Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[28] Yasushi Saito, Brian Bershad, and Henry Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 1–15, December 1999.

[29] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study. Technical report, Compaq Western Research Laboratory, November 1998.

[30] Stefan Saroiu, P. Krishna Gummadi, and Steven Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002*, January 2002.

[31] Keith Smith. *Workload-Specific File System Benchmarks*. PhD thesis, Harvard University, June 2001.

[32] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, May 1998.