An NFS Trace Player for File System Evaluation

Ningning Zhu, Jiawu Chen, Tzi-cker Chiueh, Daniel Ellard
{nzhu,jiawu,chiueh}@cs.sunysb.edu, {ellard}@eecs.harvard.edu

Abstract

File system traces have been used in simulation of specific design techniques such as disk scheduling, in workload characterization and modeling, and in identifying interesting file access patterns for performance optimization. Surprisingly they are rarely used to test the correctness and to evaluate the performance of an actual file system or server. The main reason is that up until now there does not exist a flexible and easy-to-use trace player that, given an input trace, can properly initialize the test file system and feed the trace to the test file system in such a way that respects the dependency constraint among file access requests in the trace. This paper describes the design, implementation, and evaluation of an NFS trace play-back tool called FEUT (File system Evaluation Using Traces), which can automatically derive the initial file system image from a trace, can speed up or slow down a trace play-back using temporal or spatial scaling, and features a highly efficient implementation that minimizes the CPU and disk I/O overhead during trace play-back. Experiments using a large NFS trace set show that trace-driven file system evaluation can indeed produce substantially different throughput and latency measurements than synthetic benchmarks such as SPECsfs, and FEUT's trace player is actually more efficient that SPECsfs's workload generator despite the fact that the former requires more CPU computation and disk I/O accesses.

1 Introduction

The evolution of file system design is driven by changes in workload, hardware capabilities, and user requirements regarding functionality, reliability and performance. File system traces from production file servers have been the basis for many previous studies on workload characterization, and development and evaluation of individual file system techniques. However, unlike network devices and web servers, traces are rarely used in the evaluation of complete file systems or servers. Instead, the most commonly used workloads for evaluation are synthetic macro-benchmarks and micro-benchmarks. Occasionally, performance measurements are collected on actual deployed systems [9], but this approach is unrealistic for most research file systems. The main reason that traces are rarely used in file system evaluation is the lack of a high-fidelity file system trace player that can correctly initialize an evaluated file system and faithfully observe the dependencies among trace records while playing back a file system trace. The goal of this project is to develop an NFS trace player that allows the use of traces in the evaluation of NFS servers.

The quality of a file system evaluation study is only as good as that of the input workload. An ideal file system evaluation workload should be representative of real application requirements, effective in predicting the system performance in target environments, easy to use, scalable to stress the systems under evaluation, and able to generate reproducible results. Synthetic macro-benchmarks satisfy most of these requirements except realism and predictability. Sometimes these benchmarks can be rather unrealistic and thus mislead related system research. The importance of building more realistic benchmarks in advancing the experimental computer system field has been discussed in many articles [11].

Modern file system benchmarks have made significant improvements in realism and predictability. Different benchmarks are designed specifically to capture the workload characteristics of different operating environments. Moreover, synthetic benchmarks are mostly parameterizable, making it possible to tailor the resulting workload to specific requirements. Still, it is not always possible for a synthetic benchmark to match a trace collected from a given target environment in workload realism and performance predictability for the following reasons. First, there are many time-varying and second-order or third-order factors in a workload which are very difficult (if not impossible) for a benchmark to capture. Second, because the turn-around time of developing a high-quality benchmark is often on the order of years, benchmarks usually lag behind the dynamic evolution of the workloads in the target environment. In contrast, traces are by construction representative as long as they are collected with care, and can be frequently recaptured to accommodate workload changes over time. Therefore, trace playback-based evaluation is a valuable complement to the existing benchmark-driven approach

to file system evaluation.

The major disadvantage of file traces is that it is not easy to scale them in order to stress file servers whose target performance is much higher than those from which traces were collected. This is a fundamental problem because most production file servers are set up to run at an operating point that is far less than their peak capacity in order to ensure reasonable response time. Therefore, traces collected from production file servers tend to generate less load than is required. On the other hand, with a high probability traces can capture all the other dynamic characteristics of a file system workload, including burstiness, time-varying operation mix, file access patterns, and multi-user locality, etc.

To the best of our knowledge, there does not exist a reusable trace player that can play back real traces against a file server under evaluation in the same way as standard benchmarks. We describe in this paper the design, implementation, and evaluation of an NFS trace player that allows file system researchers to evaluate their file systems or NFS servers using real traces.

The rest of this paper is organized as follows. Section 2 gives an overview of the challenges in using traces in file system evaluation. Section 3 reviews related work in trace collection, trace analysis and trace simulation. Section 4 describes the design of FEUT. Section 5 gives implementation of our NFS trace player. Section 6 presents evaluation results of the trace player. Section 7 concludes the paper with a summary of our research and gives directions for future work.

2 Overview

For trace-driven file system evaluation to become a general approach, there must be a tool to collect file traces and convert them into a standardized format and a trace player that can properly initialize the original file system image so that it play the collected traces against the file system under evaluation in a way that observes the semantic dependencies among file access commands in the traces.

2.1 Trace Collection

To build a standard trace archive, there needs to be a standard trace format, an easy-to-use trace collector, and a collection of file system traces for representative workload in different environments.

Trace collection involves many technical as well as administrative issues especially when it comes to file servers that service a large number of end users, such as privacy, service disruption, completeness, and performance impact [4]. Two main trace collection methods are instrumentation to file server kernel and snooping the network traffic of a distributed network file server. A detailed comparison between these two approaches is given in the paper [12]. Because passive snooping is less intrusive and most file servers are accessible over the network, the passive network snooping approach is the more popular of the two. A drawback of this approach is that traces collected from the network cannot be used to evaluate the client-side processing.

Ellard *et al.*have implemented a general NFS tracing tool and made this tool, along with their repository of traces, available to other researchers [6].

Traces must be represented in a standard format, so that they may be manipulated with a common set of tools. Ideally the trace format should be compact in order to reduce the disk I/O overhead of the trace player, and easy to parse in order to minimize the CPU processing cost at run time. In addition, file access requests and their responses should be paired so that it is straightforward for the trace player to ensure the evaluated file system functions correctly before starting performance measurements. To allow a trace to be played back faster than the speed at which it is collected, a dependency analysis tool is needed to identify all the dependency relationships among the trace records.

2.2 Trace Play-Back

2.2.1 File System Image Initialization

To play back requests in a file trace as if real clients are issuing them to the file server under test, the server's initial file image needs to be properly initialized so that the server can process requests in the trace. Ideally, before a trace is taken, the snapshot of the traced file server should also be taken, and later used as the initial image during trace play-back. In practice, however, this is not possible most of the time because it may cause service disruption. Therefore, it is essential to create a tool that can analyze a given trace and reconstruct an initial file system image against which the trace can be played correctly. Obviously the file system image reconstructed from a trace is not necessarily the same as the ideal snap because the trace does not always touch all the files. Moreover, the physical disk layout of a file system image reconstructed this way may be quite different from the ideal snapshot, and this difference could have significant performance implication, as file system aging has been shown to be an important factor [16]. None the less, an initial file system image reconstructed from a file represents the best one can do when the ideal snapshot is

not available, and may well be sufficiently good for most trace-based file system evaluation studies.

2.2.2 Concurrency

A major advantage of synthetic benchmarks is it is relatively easy to increase the degree of concurrency in the workload in order to stress the server under test. The common software structure of a synthetic workload generator consists of multiple processes or threads simultaneously sending requests, with requests from each process or thread operating on a disjoint directory subtree. The overall traffic load can thus be adjusted by tuning the number of processes or threads and their sending rates.

We use two methods to artificially increase the concurrency of a trace. One is to analyze the trace and identify subtraces that are independent of one another and thus can be replayed simultaneously without violating the semantics of the trace. The other is to take a trace and duplicate it as many times as the degree of concurrency requires, each time replacing all the arguments in the requests so that they operate on a separate directory tree. This option is simpler to implement and preserves the original file access timing behavior of the applications and users, but the concurrency behavior of the workload it generates is not genuine.

2.2.3 Challenges

From a convenience standpoint, it is highly desirable to be able to use a single physical client machine to emulate as many logical clients as possible and to do this at as fast as possible. To do so, a trace player must be efficient in terms of data structure design and resource usage.

Synthetic benchmarks generate file access requests in one of two ways. One way is to invoke applications such as `make`, `cp`, `grep`, `gcc`, etc., which make local file system calls, which are transformed by the local file system into NFS calls, which are sent to the server. This approach can only control application mix, but cannot directly control NFS operation mix. The other way is to directly compose and send NFS requests according to a pre-configured operation mix target. Compared with the second approach, a trace player typically incurs higher overhead for the following reasons. First, real file traces are huge and thus require substantial disk I/O and memory bandwidth during trace play-back.

For example, a typical one-day raw trace for a light NFS workload contains five million calls and responses and requires 200 MB to store in a compressed format. Second, trace records may require on-the-fly processing before they can be issued to the tested server. For example, each file object is identified by a file handle, which contains the device number, inode number and generation number. The original file handles used in the requests in the trace are not the same as the file handles generated by the tested server at run time. Therefore, the trace player needs to maintain a mapping between the original file handle and the file handle created at play-back time. Such mapping information can itself become quite large for a commercial file system containing millions of files. Finally, no trace is perfect and the trace player needs to deal with errors in the traces in a graceful way so that the play-back can continue for as long as possible. Typical NFS tracing tools can lose as many as 10% of the NFS messages during bursty traffic (although much lower rates are normal). A robust trace player needs to be able to handle various tracing errors such that their side effects can be effectively contained. This type of error processing incurs performance overhead that does not exist in synthetic workload generators.

3 Related Work

Trace-based analyses have motivated and guided file system research since the inception of the field. Analyses of how file systems are actually used has resulted in a constant progression of new optimizations and implementation techniques.

Ousterhout's 1985 trace analysis [13] set the direction for the next decade of file system research by showing that increased cache sizes would reduce read-traffic considerably, making write throughput increasingly important. This motivated work in optimizing file systems for writing, most notably LFS [15], journalling, write-clustering, and soft-updates.

The 1991 Sprite trace analysis [2] repeated this study for a distributed system and discovered that read-traffic had indeed decreased as cache sizes increased, but not as much as Ousterhout *et al.*had predicted. More importantly, it showed that the overhead of maintaining cache consistency in a distributed file system was manageable because to the relative infrequency of users actually write-sharing files.

More recent trace studies have demonstrated that file system workloads are diverse and vary widely depending on the applications they serve, and that workloads have changed over time, and raising new issues for researchers to address: Roselli *et al.*measure a range of workloads and show that files have become larger and large files are often accessed randomly, in contrast to findings from earlier studies [14]. Vogels shows that the workloads of personal computers differ from most previously-studied workloads [18], and Ellard *et al.*demonstrate that there is

a strong relationship between the names of files and their other attributes [4, 5].

As trace-based studies identify changing trends in workloads and uncover new problems for researchers to solve, it is benchmarks that are used to measure our progress. Benchmarks can be divided into two general categories: micro-benchmarks, which measure a particular aspect of file system performance (such as the time required to create a new file, scan a directory for a nonexistent file, or read one disk block from a file) and macro-benchmarks, which measure the performance of the file system at the level of an application or possibly an entire workload. Micro-benchmarks are well-suited to investigating the effects of changes to the file system because they allow isolation of these effects to the syscall layer (or below), and therefore are useful to the research community. Macro-benchmarks, on the other hand, provide a more realistic view of what the users of the system running a particular workload will actually experience. Hybrid approaches, such as hBench attempt to combine the best features of micro- and macro-benchmarks by characterizing a workload as an aggregate of simple operations, each of which can be represented by a micro-benchmark. This approach appears to work well, as long as there is a way to characterize the workload in terms of these micro-benchmarks.

In our work, we are primarily interested in workload or macro-benchmarks, so we will not dwell on the myriads of micro-benchmarks that have been developed. There are several file system macro-benchmarks in widespread use: the venerable Andrew Benchmark has been a staple of file system research papers for many years, but seems to be losing its popularity. This is probably for the best, because the development workload it represents is one that we believe is no longer very relevant – most users are unconcerned with how long it requires to compile programs. Previous analyses of our NFS traces imply that most users are primarily concerned with email [4].

The PostMark benchmark [10] attempts to simulate email, but actually only simulates a particular kind of mail workload – one with many small files. In the traces we examine, the component of the workload due to email is quite different. The methodology proposed by Elprin et al. [7] to generate email workloads by sending scripts of commands through a real IMAP server produces a more realistic workload, and can be made site-specific by creating new scripts and using a production IMAP server.

The SPEC SFS (System File Server) benchmark is an attempt to create a general-purpose benchmark for NFS servers [17]. SFS attempts to recreate a typical workload, based on a survey of traces. Unfortunately, the result does not resemble any NFS workload that we have observed. (For example, all the files created by SFS have a length of 136 KB, which is hardly typical.) Furthermore, we question whether a typical workload actually exists – each NFS trace we have examined has had unique characteristics.

File system benchmarks have evolved in parallel with advances in file system design and implementation, but they usually lag file system innovation by many years. There are two primary reasons for this. First, designing and constructing useful and accurate general-purpose benchmarks (and convincing the research community to use them) is a difficult task. The second reason follows directly from the scientific process: in order to compare two systems, it is best to run the same experiment on each. Since we cannot run today's benchmarks on the computing hardware from a decade ago, we instead benchmarks that are a decade old on our latest hardware.

One of the goals of our research is to reduce these obstacles: first, our system constructs an accurate and repeatable benchmark directly from a trace without any user intervention. Second, the resulting benchmarks are scalable: as our systems improve to the point where our current workloads are trivial, we can scale our benchmarks to the point where they will stress any system.

4 Design

FEUT consists of a pre-processing component and a traffic load generation component. The pre-processing component analyzes an input trace to derive an appropriate initial file system image, and transforms the trace into a form that is more compact and easier to use at load generation time. The traffic load generator reads in the transformed trace stored on disk, performs file handle mapping, and dispatches NFS requests in a way that satisfies inter-request dependency constraint and follows the rate requirement as specified by the user.

4.1 File System Image Initialization

In order for trace replay to succeed, we must have a file system against which to play the traces. Ideally the trace replay should begin with an exact copy of the file system from which traces were taken, but this is not always possible. Creating a snapshot of a large conventional file system may place a load on the system that is undesirable in a production environment, and there is no guarantee that the resulting snapshot will actually be consistent, because the file system may change while the snapshot is being taken. (File systems such as WAFL greatly attenuate these difficulties. [8]) The final problem is that we

must decide ahead of time when to take snapshots. For example, if we take snapshots every night at midnight, but later decide that we want to replace a trace starting at noon, the midnight trace might not be helpful.

Even if are able to acquire whatever complete snapshots we like, this only leads to another problem – a snapshot of a large server may require an equally large server to reconstruct! Ideally we would like to be able to do trace replays on relatively modest hardware.

The methods we describe in this paper allow us to infer snapshots that will allow us to replay traces

In earlier work, we describe tools that infer the directory hierarchy from an NFS trace [6]. Many of the techniques employed by these tools are not new, and date back to work by Blaze [3]. We have extended these techniques to build a tool that infers the structure of the subset of the file system that is active during the trace period. It reconstructs the file system hierarchy by observing `lookup`, `create`, `remove`, `link`, and `rename` calls and their responses, and discovers the attributes of each file via the results from `getattr`, `access`, `read`, and `write`. By scanning the trace, we can build a table of every file and directory accessed during the trace, and discover much of the file system hierarchy.

The fact that this method only discovers the active parts of the file system is both a limitation and feature. The limitation is that no information is discovered about the inactive parts of the file system, and this may have an impact on the behavior of the file system. for example, if only one file is accessed in a particular directory, the sub-snapshot for that directory will only contain that single file. If in the real system this file is only one of many in the directory, then operations on that directory may be considerably different than if there is only one.

This property has a positive aspect, however – sub-snapshots are usually a small fraction of the size of the original file system, so they can be reconstructed quickly and on small disks.

A more serious limitation is that this method of creating a snapshot does not capture the effects of that aging has had on the original file system. Most file system snapshotting methods suffer from this problem, and it is also a general problem of many file system benchmarks. There is no general solution short of examining the block layout of the original file system – and even this is not always sufficient, because the block layout may be specific to a particular file system. Hopefully, since we have long traces, the beginning part of the traces can be used as aging workload.

4.2 Trace Transformation

FEUT uses a trace record format that is designed to be sufficiently general to serve as the back-end for traces collected from different network file access protocols or from local file systems. Each trace record in FEUT is a tuple of four fields, <`timestamp`, `operation type`, `operation parameter`, `return values`>. The time at which an operation takes place is recorded in `timestamp`, whose resolution is μsec, and starts at 0 for the first trace record. The `operation type` field indicates the type of operation such as read or write, and its possible values may vary from one protocol to another. For example, NFS version 2 has 18 operations, NFS version 3 has 24 operations.

The `operation parameter` field contains the argument of the operation in question, and its value is typically a simple integer or or a string. A major type of operation parameter is file ID, which identifies the file system object involved in an operation. Inside a file system, a file object is typically identified by a device number, an inode number and a generation number. However, in a file access trace, a file can identified by its file descriptor, absolute file path, NFS file handle, etc. FEUT recognizes different forms of file IDs that point to the same object and replaces them with a FEUT-ID in the transformed trace. Since the number of unique file objects in a trace rarely exceeds millions, a 4-byte integer is sufficient for FEUT-ID. This replacement is a form of dictionary compression, which reduces the trace size.

The `return value` of a file access operation is not always available in the collected traces and may not be essential for many trace studies. But the return value is essential for FEUT to derive the initial file system image and to verify the correctness of the tested file system. In local file system traces, the return value is usually recorded together with each operation. In network file server traces, file system request and reply packets are recorded independently by the snooping software, and hence may not be next to each other in the trace. In this case, FEUT matches each request with its corresponding reply, and outputs the request together with it return value in the transformed trace.

FEUT also inserts additional information into the transformed trace to facilitate load generation. For example, the parameters of a `remove` operation are the name of the file to be removed and the file ID of its parent directory. Neither `remove` request nor its response contains the file ID of the removed file, which is needed by the load generator. The same problem exists for `rmdir` and `rename`. For all three operations, FEUT infers the file ID of the object in question from the parent's file

ID, the object name and the file system hierarchy information, and adds it to the associated record in the transformed trace.

4.3 Error Handling

Network file access traces are often not perfect, especially those long traces that are collected using network snooping for periods of weeks and months. As a result, some trace records are duplicated, and some are missing. Duplication can be easily detected, but missing packets can cause the test run to fail. In most cases, missing packets are not harmful in the sense that they will not stop the test run and thus can be safely ignored, such as `read, write, getattr, lookup`, etc. However, other missing packets can lead to abortion of test run. FEUT needs to identify these cases in the preprocessing phase and inserts the necessary packets. For example, from the following request sequence `create A; create A; remove A`, FEUT infers that there must be a `remove A` or `rename A` missing between the first `create A` and the second `create A`, and inserts one accordingly.

FEUT can also be used to verify the correctness of a file server under test. Ideally, the return value to each operation from the test file server during a trace play-back run should be the same as that recorded in the trace. If the test file server returns a different value for a file access operation, it is possible to affect subsequent requests and eventually force the test run to stop. For example, if a `create` that is followed by `write` requests to the newly created file succeeds in the original trace, but fails in the play-back run, all subsequent `write requests` can no longer be executed. Unexpected request failure during trace play-back can arise because incorrect file server implementation or transient file server problems. While these dynamic failures are not a result of FEUT implementation, FEUT still needs to handle them by skipping those operations that are affected by these failures so that the play-back run can proceed toward the end as much as possible. In contrast, synthetic file workload generators such as SPECsfs do not have this problem, because the requests it generates are created based on the the accumulative results from the set of access requests that have been successfully executed so far.

4.4 Load Generation

Before dispatching a request in a trace, FEUT first checks (1) whether all the requests that the request in question depends on are already completed, (2) whether the request's time-stamp out-runs the current time-stamp, and (3) whether the number of outstanding requests to a given server exceeds a threshold. The first constraint is absolutely required for correctness. How inter-request dependencies are identified is covered in Section 4.5. The second is optional and is mainly for trace scaling as discussed in Section 4.6. The third is also optional and is included mainly to avoid overloading the test file server. If a file server is overloaded, it tends to have much low throughput. Without the third check, eventually subsequent requests will be stopped if they depend on previous uncompleted requests, or if the network buffer is exhausted if the connection between client and file server is built upon TCP. Third check serves as an overload protection, which ensures that a file server being evaluated always operates at a realistic operating point.

A trace play-back is said to run at full speed if only the dependency constraint check and maximum outstanding request check is performed every time a request is dispatched. The initial part of a trace play-back run is typically used to warm up the test file server's cache. At the warm-up stage, requests are dispatched at full speed.

4.5 Dependency Analysis

If an operation P appears before another operation Q in a trace, P and Q operate on some common file object(s), and at least one of them modifies the common objects(s), Q must to be executed after P has been finished. In this case, Q is said to depend on P, which is represented as `P->Q`. A request in a trace can be dispatched only when all its dependees are completed. Here a file system object can be a regular file, directory, symbolic link or special device node. In NFS, modification to a regular file is through such operations as `create, write, commit, setattr` and `remove`, and access to a regular file is through operations such as `read, getattr, access, lookup`, in addition to the modification operations. Note that the `create` of a file is considered as modification to a file, because it has to be serialized with all other accesses `read, write` to this file. For similar reason, `remove` is considered as modification to a file too.

Modification to a directory is by adding/deleting/renaming file system objects under this directory and `setattr` operation, and access to a directory involves `readdir` and `lookup` of file system objects under the directory. The operation `readlink` is considered as access to the symbolic object. Operations that access overall file system information such as `fsstat` or `fsinfo` are ignored in the dependency analysis, which means that these operations do not depend on any other operation and no operation

depends on them. Table 1 shows how file system objects used in common file system operations are considered "accessed" or "modified." Note that if a `lookup obj` operation succeeds, the object's handle will be returned and we consider this `lookup` request accesses both the parent and the returned `obj`; otherwise we consider this `lookup` request only accesses the parent. Similarly, the [obj2] in `rename` is the target object which may or may not exist before the rename. If the object exists, it will be deleted as result of `rename`.

Dependency constraint determines the upper bound on the degree of concurrency available in a trace. That is, a trace player can maximize the number of concurrent requests that can be dispatched by looking as far ahead into the trace as possible to identify the independent requests, in the extreme case the entire trace. The degree of concurrency possible depends on the granularity of dependency analysis. FEUT currently uses file system objects that can be uniquely identified by their file ID as the analysis granularity. Higher concurrency is possible if FEUT chooses finer analysis granularity such as file attribute, file block, and directory entry, which can be identified by a combination of file ID with flags, logical block number and name string, respectively.

4.6 Scaling

The scalability of a workload generator is defined as its ability to stress file servers with a wide range of throughput characteristics. In the case of FEUT , if a trace runs at 100 requests/sec when it is collected, how can FEUT scale it up to stress a 10000 requests/sec server, and scale it down to test a 10 requests/sec server? In the process of scaling, it is important to preserve the file access characteristics of the original trace as much as possible, as there is no point of using a trace-driven approach if all the characteristics important to file system evaluation are destroyed in the scaling process.

The degree of concurrency required to stress a file server is determined by the product of its throughput T and latency L, $T * L$. Because a high-performance file server typically reaches its peak throughput at high processing latency, high concurrency is needed to stress a high-end file server. According to the benchmark result published by SPECsfs for the *sfs97_R1.v3* benchmark, the SpinServer 3300 (6-node scalable cluster) from Spinnaker Networks Inc. has the the highest maximum concurrency requirement of 752, with a peak throughput of 131930 requests per second and a latency of 0.0057 second at peak throughput, where as RainFiler (Failover,1-Node,4x36,TCP) from Traakan Inc. has the lowest maximum concurrency requirement of 9, with a peak throughput of 1957 requests per seconds and a latency of 0.0046 second.

There are two ways to speed up a trace. The first approach, called *spatial scale-up*, generating multiple evaluation loads on disjoint working directory. The multiple loads can be homogeneously loads the same trace or heterogeneous loads from different traces, The overall concurrency of multiple loads is the sum of the concurrency of individual load. Spatial scale-up preserves the timing relationships among the requests in the original trace(s). For homogeneous loads, random delay is introduced for the starting time of each load to avoid the all the loads encounter bursty operation simultaneously. The range of delay time is configurable but should be less than the warm-up time. The total working file set size is the sum of of all disjoint directories, and thus grows linearly with respect to the number of loads and degree of concurrency. This approach is simple to implement, Duplication is also used in most synthetic file system workload generators, such as SPECsfs.

The second approach, called *temporal scale-up*, it simply speeds up a trace by dividing the time-stamps by a speed-up factor and playing the trace according to the new time-stamps. If the evaluated file system is slower than the original file server, operations may be delayed due to traffic burst. Trace play-back monitors the delayed packets and gives periodical statistics about the average delay time, total number, and percentage of delayed packets. Temporal scale up preserves the burstyness, total working file set size, locality, and order characteristics of file access requests in the trace.

There are also two ways to slow down a trace. The first approach, called *temporal scale-down*, is the opposite of *temporal scale-up*. It simply slows down a trace by multiplying the time-stamps by the slow-down factor. The second approach, called *spatial scale-down*, decomposes a trace into subtraces according to client IP address, group ID, user ID or working directories. Each subtrace resulting from such a decomposition may not be self-contained if files are being shared among clients, groups or users. For example, if dir-A is shared between user1 and user2, user1 deletes file-B under dir-A, and user2 creates file-B under dir-A, then the subtrace for user1 will contain only the `delete` operation and the subtrace for user2 will contain only the `create` operation. As a result, both subtraces may not be able to be played back correctly. FEUT solves this problem by running each resulting subtrace through the trace preprocessing step, which handles such missing requests in the same way as missing packets in the original trace. The *temporal scale-down* approach preserves the total working file set size while the *spatial scale-down* decreases the total working file set size.

File System Operation	Modified	Accessed
read/readdir/getattr/readlink obj		obj
write/setattr/commit obj	obj	obj
lookup parent, name([obj])		parent, [obj]
create/mkdir parent, name(obj)	parent, obj	parent, obj
remove/rmdir parent, name([obj])	parent, [obj]	parent, [obj]
symlink parent, name([obj]), path	parent, obj	parent, obj
rename parent1, name1, parent2, name2([obj2])	parent1, parent2, [obj2]	parent1, parent2, [obj2]

Table 1: This table shows how file system objects used in common file system operations. are considered "accessed" or "modified." [obj] means that the object may not exist and the operations might return failure.

If a file system trace is dominated by a large number of request streams each of which operates an independent file system object sets and generates relatively light access load, spatial scaling is more appropriate in speeding up or slowing down the trace. A typical example of such a workload is a software development environment where file accesses are interleaved with user think time and CPU/networking activities. If a file system trace is dominated by a small number of request streams each of which generates relatively heavy file access load, temporal scaling is more appropriate in speeding up or slowing down the trace. A typical example of such a workload is a supercomputer cluster on which multiple jobs are executed concurrently on disjoint sub-clusters, which share a single file server.

Most real-world file system workloads are a mix of these two type of workloads. One needs to understand the nature of the traces and the target workloads, and decide the best scaling approach based on that understanding. These two scaling approaches can also be combined together. For example, if the required speed-up factor is 12, it can be achieved by a spatial scale-up factor of 3 and a temporal scale-up up factor of 4.

5 Implementation

A file system workload generator must determine what requests to dispatch and when to dispatch them. To minimize the run-time overhead of NFS request generation, FEUT pre-processes a trace to make it more compact and easier to parse, and at run time overlaps trace reading from the disk with request generation to improve the concurrency.

5.1 NFS Trace Pre-processing

The FEUT preprocessing tool is implemented in Perl and consists of two passes. The first pass scans the trace and and infers a *hierarchy map* between each file system object that appears in the trace and its absolute path-name and lifetime. Each file system object is uniquely identified by *FILEID*, a 4-byte integer.

The format of each *hierarchy map* entry is <*FILEID, PATHNAME, LIFETIME-START, LIFETIME-STOP*>. Entries in the map can be accessed directly by either the *FILEID* or <*PATHNAME, TIMESTAMP*> tuple. A *FILEID* usually corresponds to only one matching entry, but multiple entries may exist due to hard links, which permit a single file to have more than one name in the file system hierarchy. A <*PATHNAME, TIMESTAMP*> tuple should map to at most one entry, and the matching entry should have the same *PATHNAME* and the *TIMESTAMP* should be between the entry's *LIFETIME-START* and *LIFETIME-STOP*. If a file system object exists before the trace collection starts, its *LIFETIME-START* is set to 0 and if a file system objects exists when the trace collection ends, its *LIFETIME-STOP* is set to infinity. The initial file system image for a trace corresponds to those entries in the *hierarchy map* whose *LIFETIME-START* is 0.

In the second pass, the FEUT pre-processing tool converts each file handle into its corresponding *FILEID*, combines each NFS request with its associated return value, eliminates unnecessary information in the original trace, fills up the deleted object's *FILEID* for `remove`, `rmdir` and `rename` operations, and properly inserts requests to accommodate missing packets. While still in a human readable text format, the size of the transformed trace is typically about 10% of that of the original trace.

The current FEUT pre-processing tool implementation is not incremental. Considering the large trace size (>100 GBytes), incremental pre-processing is an important feature. The main limitation is that the current prototype always builds the *hierarchy map* from scratch. One can use a standard check-pointing mechanism to protect the pre-processing phase from software/hardware crashes.

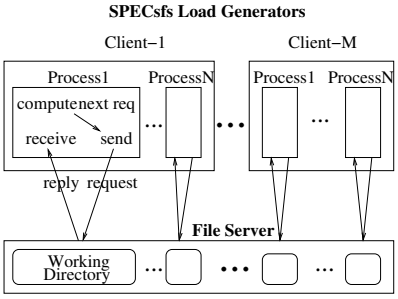

Figure 1: **SPECsfs Software Structure**

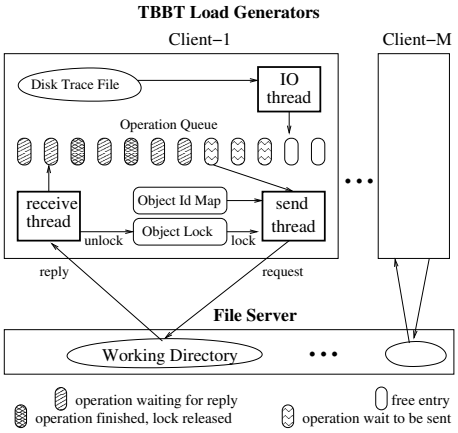

Figure 2: **FEUT Load Generator Software Structure**

5.2 NFS Workload Generation

The FEUT NFS workload generator is a user-level utility written in C, and adopts two techniques similar to SPECsfs [17] to improve its efficiency. First, because the input NFS trace is typically collected via network snooping, FEUT bypasses the NFS client and sends NFS requests directly to the tested file server using user-level RPC. The advantage of this approach is that the workload is not effected by the NFS client-side processing.

The second technique is to run the trace player on multiple machines to collaboratively generate a higher workload, and to collect the resulting performance measurements from them, all in a way transparent to the user.

The software architecture of FEUT , however, is different from SPECsfs. As shown in Figure 1, the workload generator of SPECsfs on each client machine uses a multi-process software architecture, with each process dispatching NFS requests using synchronous RPC. In contrast, FEUT uses a 3-thread software structure, as shown in Figure 2. This is more efficient because it reduces the context switching and scheduling overhead. The *IO thread* continuously reads trace records into a cyclic memory buffer called the *operation queue*. The *send thread* and *receive thread* send NFS requests to and receive their replies from the NFS server under test using asynchronous RPC requests. The size of the *operation queue* should be several times larger than the theoretical concurrency bound of the input trace to ensure that the *send thread* is always able to find enough concurrent requests at run time.

The *send thread* determines whether an NFS request in the input trace is ready to send by checking its dependency constraint, temporal/spatial scaling constraint, and the maximum number of outstanding requests constraint. Enforcing the temporal/spatial scaling constraint and the maximum number of outstanding request constraint is relatively straightforward. However, checking the dependency constraint is more complicated. We have explored two alternative methods for managing the dependency constraint: *object locking* and a *dependency list*.

The *object locking* approach is used in the current prototype and is illustrated in Figure 2. Before dispatching an NFS request that modifies state on the server, the *send thread* acquires a write lock on the object(s) modified by that request. Before dispatching a read-only NFS request, the *send thread* acquires a read lock on the object(s) it operates on. For example, a request to create a new file in a directory will lock the directory object, so that another request that accesses the directory (for example, a `lookup` request) will be blocked.

When the *receive thread* receives an NFS reply, it releases any locks held on the object(s) that the corresponding NFS request acquires. Each lock is just a flag associated with each file system object in *hierarchy map*, and hence very light-weight. Because one operation may need to lock multiple objects, to avoid deadlock, the *send thread* will acquire the locks associated with one NFS requests only if all locks are available. The *send thread* itself is non-blocking with respect to lock acquisition. It keeps polling over the *operation queue* to check which request can be dispatched. If there is no request can be dispatched, *send thread* waits in blocking mode for *receive thread* to receive at least one reply, and then checks the *operation queue* again.

The *dependency list* approach requires the preprocessing tool to identify all the previous requests that an NFS request directly depends upon and check that dependency list before issuing that request. With *dependency list* information, run-time dependency checking is fairly simple: before dispatching an NFS request Q, the *send thread* checks whether all requests in its dependency list have already been completed. Note that only direct dependency needs to be recorded in the dependency list. That is, if Q depends on P and R depends on Q, P does not need to appear in R's dependency list. Most NFS requests only access one or two

file system objects, and therefore have a fairly small dependency list. Assume each entry in a dependency list is represented by an integer of 4 bytes. The average dependency list is between 4 to 8 bytes, which is not significant considering that the on-disk and in-memory representations of each NFS request cost around 60 bytes and 100 bytes, respectively. However, dependency list may vary in size from request to request, and therefore requires a flexible data structure. For example, in the request sequence `write a; read a; read a; read a; read a; write a`, each read operation depends only on the first write, whereas the last write depends on all 4 read operations (because if it is issued before the last read completes, it might obliterate the value that would be read by one or more of the reads).

5.3 Running the Trace Player

Similar to SPECsfs [1], the FEUT trace player also has an initialization stage, a warm-up stage, and a measurement collection stage. The initialization stage fills up the test file server with a proper initial file system image. The warm-up stage is meant to age the test file system and warm its file cache by playing the initial part of the trace. As future work, advanced aging techniques such as those described by Smith etal. should be incorporated [16]. FEUT allows users to control the scaling of a trace through the following parameters, *trace/sub-trace selection*, *spatial scaling factor*, *temporal scaling factor* and *maximum outstanding request number*. At the end of a play-back run, FEUT reports various statistics about the throughput, latency, operation mix, initial file system size and accessed data set size, as in SPECsfs. FEUT also provides throughput and latency evolution over time during a trace play-back run to give users a better insight into the workload burstiness and temporal variation inherent in the input trace. For example, in a real workload, different hours of a day and different days in a week often exhibit very different workload characteristics [4].

6 Evaluation

6.1 Methodology

In this section, we evaluate the value, implementation efficiency and scalability of FEUT. The value of a trace-based approach for file system evaluation lies in the fact that it can capture a particular site's workload characteristics and at the same time produce sufficiently different results than synthetic benchmarks such as SPECsfs [17]. The implementation efficiency of FEUT limits the kinds of file servers it can test. Because synthetic workload generators require relatively less CPU and disk overhead, their performance also serves as the reference case for FEUT. The scalability of FEUT concerns its ability to scale up and down the generated workload based on the input trace to test file servers rated at a wide variety of performance classes.

The traces we used in this study were collected from the EECS NFS server (EECS) and the central computing facility (CAMPUS) at Harvard [4]. The EECS workload is dominated by metadata requests and has a read/write ratio of less than 1.0. The CAMPUS workload is almost entirely email and is dominated by read requests. The EECS trace and the CAMPUS trace grow by 2GBytes and 8 GBytes per day, respectively.

We use the collected traces to drive two NFS servers, one is the Linux-based NFS server, and the other is a repairable NFS server called RFS [19], which augments a generic NFS server with fast repairablity, but does not change the network file system protocol or network file access path. The file server machine that hosts NFS and RFS servers is a server machine with 1.5-GHz Pentium 4 CPU, 512-MByte RAM, and one 40-GByte ST340016A ATA disk drive with 2MB on-disk cache. The OS is Redhat Linux 7.2 with NFSv3 enabled. FEUT's trace player runs on a client machine that has a 1.8-GHz Pentium 4 CPU, 512-MByte memory, and 20 GB disk.

6.2 Value of FEUT

A trace must be valid before it can be of value to file system evaluation. One notion of trace validity is that a trace faithfully capture the workload characteristics of a particular site. We assume traces are automatically valid in this sense if the traces are collected over a sufficiently long period of time. The other notion of validity is that a trace itself does not miss out many packets during the tracing period. Packet loss during trace collection lead to packets that can not be paired with their replies or requests, or gap in RPC message ids. Most of the Harvard traces have a packet loss ratio of 0.1% - 10%.

The size and structure of initial file system image directly effects the performance results. Whenever initial disk image snapshot is not available, FEUT rely on *hierarchy map* to generate a close-enough file system image. Any <parent, child> relation discovered by preprocessing is correct but they are usually incomplete. As a result, the *hierarchy map* will be multiple isolated subtrees as opposed to one big tree starting from root in the original file system. Large number of subtrees indicating problems about missing <parent,child> relation-

	original load		scale-up		peak load	
Operation	**SPEC**	**FEUT**	**SPEC**	**FEUT**	**SPEC**	**FEUT**
Throughput	33	30	189	180	1231	1807
getattr	5.1	0.6	0.9	1.5	2.1	0.7
lookup	2.9	0.9	0.8	2.0	2.0	1.2
read	9.6	3.1	5.3	4.8	5.4	4.7
write	9.7	2.2	4.4	3.8	4.6	2.5
create	0.5	0.7	0.7	0.9	17.3	0.7

Table 2: NFS latency/throughput for EECS trace at Oct 21, 2001.

	original load		scale-up		peak load	
Operation	**SPEC**	**FEUT**	**SPEC**	**FEUT**	**SPEC**	**FEUT**
Throughput	32	30	187	180	619	1395
getattr	4.0	0.7	2.2	1.2	3.2	0.8
lookup	4.4	0.7	2.8	1.3	2.6	1.0
read	10.8	3.3	8.4	4.1	18.1	4.9
write	11.6	5.4	7.4	4.0	11.1	2.8
create	0.7	1.0	5.1	1.3	16.3	1.2

Table 3: RFS latency/throughput for EECS trace at Oct 21, 2001.

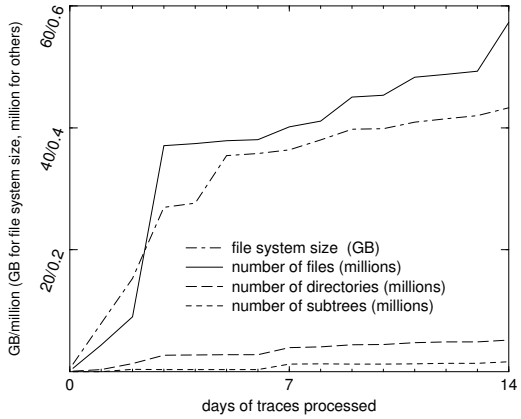

Figure 3: **Discover Initial File System Image**

ship. However, small number of subtrees is not necessarily good, because a whole subtree in the original file system can be completely missing. Figure 3 gives the number of subtrees, number of file, number of directories, and total file system size discovered by preprocessing traces of different size, from one day to two weeks. The trace used in this study is from Oct 15 (Monday), 2001 to Oct 29, 2001. The total file system size on EECS server is 400 GB and 42 GB is discovered in 14 days. At the end of the period, the number of subtrees has stopped growing and remains near constant, most of them corresponds to one user's home directory. The total number of directories also remains near constant. This probably shows that directories are accessed more regularly and easy to be discovered. On the contrary, regular file discovery shows different pattern. Even after two weeks, every day there is new discoveries about old files. We will have to conduct more experiment to find the converge point, which can be either the 400 GB upper limit, or some other number which represents the active portion of the file system.

To compare the evaluation results from a trace player such as FEUT and from a synthetic workload generator such as SPECsfs, we chose a Harvard trace collected on Oct 21, and tried to tune the SPECsfs benchmark to match the trace's workload as match as we could, including the total file set size (6GB), total number of files (24000), total number of directories (8000), the percentage of each operations, the accessed file set percentage (30%) and the append write vs. overwrite ratio (80% overwrite). We also changed the SPECsfs source code so that its file size distribution matches the file size distribution in the Oct 21 trace. The maximum throughput of the Linux NFS server under SPECsfs is 1231 requests/sec, and is 1807 requests/sec under FEUT. The difference is a non-trivial 31.8%. In terms of per-operation latency, Table 2 lists the latency of fiver different operations under the original load (30 requests/sec), a scale-up speed of 7, and the peak load. Again the per-operation latency numbers are quite different in many cases.

We carried out the same experiment using the RFS server as the target. The maximum throughput of the RFS server under SPECsfs is 619 requests/sec, and is 1395 requests/sec under FEUT, more than a factor of two difference. Table 3 lists the latency of five differ-

ent operations under the original load (30 requests/sec), a scale-up speed of 6, and the peak load. Again there is no obvious correlation between the per-operation latency numbers from SPECsfs and from FEUT.

Another experiment is conducted using Oct 22 (Monday) trace. The initial file system size is similar (around 6 GB) but there are twice as many file (40000). The Oct 22 trace is dominated by metadata operation (80%) while Oct 21 has substantial read/write operations (60%). The SPECsfs configuration is again tuned differently to match the Oct 22 trace. The result is shown in Table 4

If the Oct 21/22 traces are indeed representative of the weekend/weekday workload at the Harvard site, the results from the above experiments suggest that performance results derived from synthetic benchmarks such as SPECsfs may deviate from the actual results by a substantial margin. Therefore, trace-based file system/server evaluation indeed has its value as long as the process can be largely automated.

6.3 Implementation Efficiency

FEUT's pre-processing tool is implemented in Perl, and is able to process 2.5 MBytes of trace or 5000 requests per second. The FEUT pre-processing time is nearly linear with respect to the trace size, and is about 30 minutes for one day worth of the EECS trace, and 2 hours for one day worth of the CAMPUS trace.

The efficiency of FEUT's trace player is determined by its disk I/O, CPU and memory access overhead. The disk bandwidth requirement of FEUT's trace player is fairly small. Each trace entry costs less than 100 bytes. According to the most recent throughput results published by SPECsfs, the throughputs of file servers range from 2000 to 300000 requests/sec, which means that the *I/O thread* needs between 0.2 and 30 MBytes/sec disk bandwidth to read traces from disk. Given that the I/O thread is the only thread that accesses the disk during trace play-back and trace access involves mostly large sequential reads, it is unlikely that the disk could become the performance bottleneck.

The CPU load of a trace player comes from the send thread, receive thread and the network subsystem insider the OS. When the Linux NFS server runs under a trace at peak throughput (1807 requests/sec), the measured CPU utilization and network bandwidth consumption for the FEUT trace player are 15% and 60.5 Mbps. When the same Linux NFS server runs under a SPECsfs benchmark at peak throughput (1231 requests/sec), the measured CPU utilization and network bandwidth consumption for the SPECsfs workload generator are 11% and 37.9 Mbps. These results suggest that FEUT's trace player is actually more efficient than SPECsfs's workload generator, despite the fact that trace play-back requires additional additional disk I/O for trace reads, and additional CPU overhead for dependency detection and error handling. FEUT out-performs SPECsfs because FEUT's trace player uses only 3 threads, whereas SPECsfs uses multiple processes and thus incur excessive context switching and process scheduling overhead.

The memory consumption of the trace player mainly comes from the following data structures:*hierarchy map*, *object lock*, and *operation queue*. Practically speaking, the number of files on a file server rarely exceeds 10 million, combining *hierarchy map* and *object lock map* together with their hashing structure results in around 100 bytes per entry. The total size of *object id map* and *object lock map* is thus no more than 1 GByte.

The *operation queue* stores all the requests which is being processed by the file server or which is considered as candidate for next dispatch. The size of the *operation queue* sets an upper limit for the play-back concurrency and is called *look-ahead window*. A small *look-ahead window* size artificially reduce the workload concurrency and may have negative effect on file system throughput being measured. The *look-ahead window* should be proportional to the workload concurrency as computed by throughput*latency, and sufficiently large to accommodate burst of operations which all depend on each other. Section 6.4 described a simulation study for this purpose. According the simulation result, *look-ahead window* size of 4000 is enough for EECS with a concurrency value of 80, and 8000 is good enough for CAMPUS trace with a concurrency of 160. In summary, for an workload like EECS and CAMPUS trace, a *look ahead* window 50 times larger than the trace concurrency is enough.

The small *look ahead window* means that the *operation queue* will not occupy too much memory. In Section 4.6, we mentioned that the maximum concurrency required by fastest NFS file server available is 752. Each entry in the the *operation queue* is about 200 bytes, therefore the entire *operation queue* costs no more than 752*200*50 = 7.5 M Bytes .

6.4 Trace Scalability

Scaling a trace spatially up and down does not have any particular limitations. Scaling a trace down temporally also is relatively straightforward. However, scaling a trace up temporally requires there is sufficient concurrency in the trace. To study the maximum concurrency that can be generated from a trace, we conducted a simulation in which a request can be dispatched if all the requests it depends on are completed. In this mode, there

	original load		scale-up		peak load	
Operation	**SPEC**	**FEUT**	**SPEC**	**FEUT**	**SPEC**	**FEUT**
Throughput	16	15	191	187	2596	4125
getattr	4.7	0.5	0.7	0.7	1.02	0.7
lookup	2.8	0.6	0.5	0.8	1.01	0.6
read	10.3	2.1	19.7	3.1	7.4	4.2
write	7	1.0	6.3	1.2	3.8	3.0
create	0.5	0.9	1.2	0.5	7.9	0.7

Table 4: NFS latency/throughput for EECS trace at Oct 22, 2001.

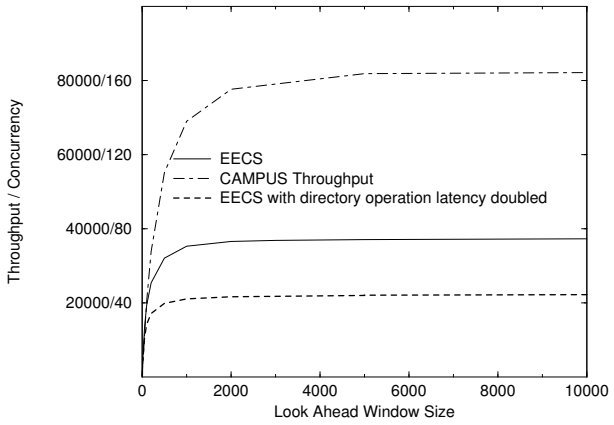

Figure 4: **Discover Trace Concurrency**

are two factors that limit the maximum concurrency: the *look-ahead window* in which future requests are examined during the simulation, and the per-request latency at the server. For the per-request latency, we used the latency numbers in Table 2 and Table 4. Figure 4 shows the correlation between the maximum throughput that can be generated and the amount of look-ahead. The dashed line shows the effect of multiplying the latency of each directory request by a factor of 2.

In general, the larger the *look-ahead window*, the more concurrency can be extracted from the trace, and the heavier load the trace can generate. When the *look-ahead window* is small, the trace's maximum throughput and concurrency is limited by the *look-ahead window*. However, beyond a certain size, dependency among requests in the trace is the dominating constraint.

Concurrency can be calculated from (throughput*latency). Because the average latency used in this simulation is about 2 msec, the maximum concurrency is 80 for the EECS trace and 160 for the CAMPUS trace. The simulation results show that even in an originally lightly loaded EECS trace (30 requests/sec) and a modest *look-ahead window* size (4000), there is enough concurrency to drive a file server with a performance target of 10000 requests/sec.

7 Conclusion

The prevailing practice of evaluating the performance of a file system is stressing the test file system using synthetic benchmarks. While sophisticated benchmarks do incorporate characteristics of actual traces and are capable of generating file access workloads that are representative of the load in real operating environments, they rarely are capable of fully capturing the time-varying and second-order effects of the workload at a specific site. In this paper, we advocate a trace-driven file system evaluation methodology, in which one evaluates the performance of a file system/server by driving it with real traces and measuring its resultant latency and throughput, and describe an NFS trace play-back tool called FEUT that is designed to support this methodology. FEUT addresses most if not all of the trace-driven workload generation problems, including automatic derivation of initial file system from a trace, satisfying the dependencies among requests in the trace during play-back, scaling a trace to a play-back rate that can be higher or lower than the speed at which the trace is collected, graceful handling of errors in trace collection and in incorrect behavior of test file system/server, and efficient implementation that allows a single trace play-back machine to drive a wide range of high-performance file servers. The result is a flexible and easy-to-use NFS trace player that is shown to be actually more efficient than SPECsfs, a state-of-the-art synthetic file system workload generator.

In addition to being a useful tool for file system researchers, perhaps the most promising application of FEUT is to use it as a site-specific benchmarking tool for comparing competing file servers. That is, one can compare two or more file servers for a particular site by first collecting traces on the site, and then testing the performance of each of the file servers using the collected traces. Evaluation results derived from such a procedure should be as real as they can get, assuming the traces collectively are representative of that site's workload.

Although the current prototype can only play back NFS traces, its internal trace format is designed to serve

as a common back-end for traces based on network file access protocols other than NFS, such as SMB, CIFS, and AFS. We plan to develop a converter that can translate CIFS traces collected from a SAMBA server into FEUT's internal format, and use FEUT to play back the resulting trace against a Windows-based CIFS server.

References

[1] *System File Server Benchmark SPEC SFS97_R1 V3.0*. Standard Performance Evaluation Corporation. (http://www.specbench.org/sfs97r1/).

[2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Monterey, CA, October 1991.

[3] Matthew A. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, San Fransisco, CA, January 1992.

[4] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[5] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. The Utility of File Names. Technical Report TR-05-03, Harvard University Division of Engineering and Applied Sciences, 2003.

[6] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Seventeenth Annual Large Installation System Administration Conference (LISA'03)*, pages 73–85, San Diego, CA, October 2003.

[7] Nicholas Elprin and Bryan Parno. An Analysis of Database-Driven Mail Servers. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*, San Diego, CA, USA, October 2003.

[8] D. Hitz et al. File system design for an nfs file server appliance. In *USENIX winter 1994 conference*, pages 235–246, Chateau Lake Louise, Banff, Canada, 1994.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

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

[11] J. Mogul. Brittle Metrics in Operating Systems Research. In *The Seventh Workshop on Hot Topics in Operating Systems: [HotOS-VII]: 29–30 March 1999, Rio Rico, Arizona*, pages 90–95, 1999.

[12] Andrew W. Moore. Operating System and File System Monitoring: a Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques. Master's thesis, Monash University, 1995.

[13] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA, December 1985.

[14] Drew Roselli, Jacob Lorch, and Thomas Anderson. A Comparison of File System Workloads. In *USENIX 2000 Technical Conference*, pages 41–54, San Diego, CA, 2000.

[15] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[16] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[17] SPEC SFS (System File Server) Benchmark, 1997. http://www.spec.org/osg/sfs97r1/.

[18] Werner Vogels. File System Usage in Windows NT. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, SC, December 1999.

[19] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In *The International Conference on Dependable Systems and Networks*, June 2003.